11/02/2011

TFC

Treball Fi de Carrera

Anàlisi d'estratègies d'accés a dades

Memòria

Estudiant: Sergi Tarrat i Labaila
Correu: starrat@uoc.edu
Consultor: Jordi Sánchez Cano
Carrera: ETIG
Àrea: Microsoft.NET

ÍNDEX

1 Descripció del projecte

1.1 Justificació

Tot i que quan vàrem escollir aquest projecte no havíem sentit a parlar mai encara del concepte de Mapeig Objecte Relacional (ORM) i no ens fèiem per tant massa al càrrec de la justificació que podia tenir aquest treball (tret del propi coneixent de les eines de la plataforma de programació .NET de Microsoft) a mesura que hem anat estudiant i coneixent els conceptes bàsics del model, hem descobert la seva importància real i la justificació, per tant, de projectes com aquest.

Per una banda, avui dia la gran majoria d'aplicacions es desenvolupen sota el paradigma de l'orientació a objectes (OO) i aquestes cada cop són més i més complexes. Per altra banda, no ha triomfat encara al mercat cap BBDD orientada a objectes. En el moment d'implementar la persistència és quan realment ens trobem amb el problema que representa l'haver d'anar convertint contínuament (al guardar, afegir, actualitzar, eliminar i recuperar les dades) entre els models conceptual i relacional.

És molt important doncs sota aquesta conjuntura, que existeixin models i implementacions tecnològiques que permetin desvincular al programador del problema que significa la comunicació entre els dos móns. Ens ha sorprès molt gratament la bona idea que representa el model ORM i que ja existeixin algunes tecnologies que permetin la seva implementació, tot i que com veurem més endavant en general encara no estan suficientment madures i de vegades pot ser més complexa la solució escollida que el propi problema a solucionar.

1.2 Abast

La pretensió bàsica d'aquest treball de fi de carrera ha consistit en l'anàlisi de diverses estratègies d'accés a dades usades en l'entorn de desenvolupament d'aplicacions amb la plataforma Microsoft .NET, entrant especialment a detall en les darreres tecnologies que implementen el model ORM.

Bàsicament hi ha hagut dues grans etapes que han coincidit amb els lliuraments de les PAC 2 i 3:

- A la PAC2 hem desenvolupat la part teòrica del projecte, basada en l'estudi, recerca i documentació de les diferents estratègies seleccionades, aprofundint en les seves funcionalitats i afecció sobre la resta de capes del sistema.
- La PAC3 en canvi, ha estat enfocada des de una vessant pràctica i ha consistit en el desenvolupament d'una petita aplicació d'accés a dades on hem usat cadascuna de les estratègies estudiades a la PAC anterior, més ADO.NET, que hem afegit sobre la marxa per a permetre la comparació entre el model tradicional que aquesta darrera tecnologia representa i el nou model abanderat per al patró ORM.

1.3 Objectius

Els objectius que ens hem marcat alhora de desenvolupar aquest treball són els descrits a continuació:

1. Descripció general de la necessitat de les estratègies d'accés a dades, i particularment d'un model que simplifiqui la distància existent entre els models conceptual i relacional.
2. Recerca, estudi i coneixement de les principals tecnologies ORM existents i l'afecció d'aquestes sobre la resta de capes del sistema.
3. Elaboració del treball pràctic, que ha consistit en la implementació d'una petita aplicació d'accés a dades, utilitzant cadascuna de les diferents tecnologies estudiades: ADO.NET, Entity Framework, NHibernate i LINQ to SQL.

1.4 Enfocament metodològic

En primer lloc es va presentar el problema a resoldre i la necessitat de l'existència d'implementar alguna estratègia d'accés a dades, per a tot seguit elaborar la planificació del projecte (PAC 1).

El següent pas va ser iniciar el treball de recerca usant els mètodes habituals (Internet i bibliografia especialitzada d'aquesta temàtica) on es varen estudiar les tècniques seleccionades i es va aprofundir en el seu coneixement, descobrint els trets característics de cadascuna d'elles. Un cop completat l'estudi es va procedit a elaborar l'informe tècnic (PAC 2).

A continuació es va desenvolupar una petita (però complexa) aplicació per a conèixer pràcticament cadascuna de les estratègies estudiades (PAC 3).

Finalment s'ha procedit a l'elaboració d'aquesta memòria i la presentació del projecte.

1.5 Equips i tecnologies emprades

En la realització d'aquest projecte s'han usat els següents equips i tecnologies:

1.5.1 Maquinari i programari

- PC de Sobretaula AMD Athon 64 X 2 Dual Core Processor 4200 a 2,21Ghz i 1.00Gb de memòria RAM.
- Microsoft Windows XP Professional service pack 2.
- Microsoft SQL Server 2008.
- Microsoft .NET Framework 3.5 service pack 1.
- Microsoft Visual Studio 2008 service pack 1.
- Microsoft Office 2003 (Word i Project).

1.5.2 Tecnologies

Totes les tecnologies de programari emprades en el desenvolupament del projecte són propietàries de Microsoft tret de NHibernate, programari lliure distribuït sota els termes de la LGPL (Llicència Pública General Menor de GNU).

1.6 Planificació inicial

La planificació del treball ha vingut marcada per les dates de lliurament de cadascuna de les PAC, que han estat les següents:

- **PAC 1.** Pla de treball. Del 21/09/2010 al 04/10/2010. Descripció general i planificació del projecte.
- **PAC 2.** Estudi, recerca i documentació. Del 05/10/2010 al 01/11/2010.
- **PAC 3.** Anàlisi, disseny i implementació. Del 02/11/2010 al 20/12/2010.
- **Final.** Memòria i Presentació. Del 21/12/2010 al 10/01/2011.
- **Debat virtual.** Defensa del treball. Del 24/01/2011 al 27/01/2011.

En el diagrama de Gannt podem veure en detall la planificació del projecte que es va preveure inicialment:

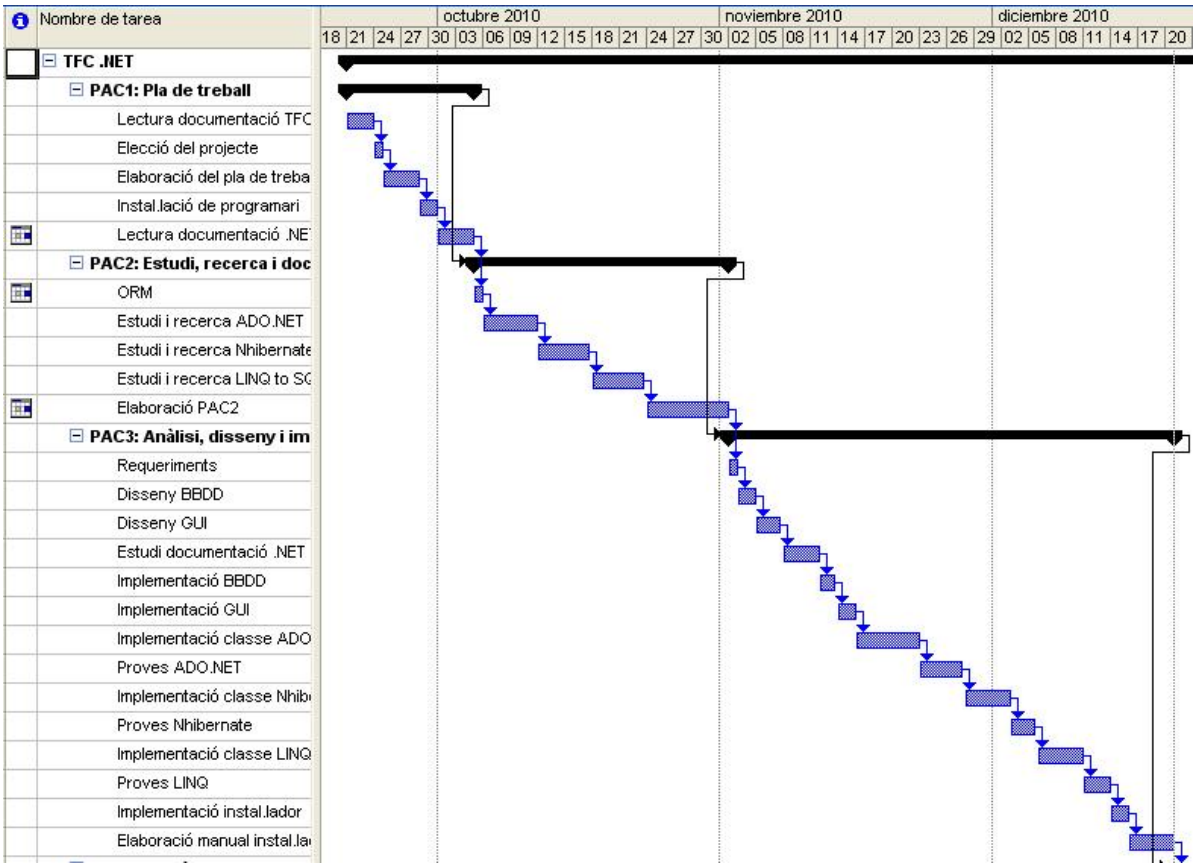


Figura 1. Planificació inicial

1.7 Planificació real

Sobre les dades pràcticament no hi ha hagut cap variació a remarcar i s'han pogut assolir tots els lliuraments a temps sense incidències. El que si que ha canviat lleugerament és el contingut. Tot seguint expliquem aquestes lleus desviacions.

Durant l'elaboració del pla de treball a la PAC 1, inicialment es va pensar en estudiar les tres tecnologies següents: ADO.NET, NHibernate i LINQ to SQL.

Posteriorment durant l'estudi realitzat a la PAC2 vàrem descobrir el model ORM i les seves possibilitats, motiu per al qual vàrem decidir deixar ADO.NET (que per altra banda és una tecnologia ja prou coneguda) solament com a introducció del model relacional tradicional, i centrar l'estudi en 3 frameworks que implementessin el model ORM. En aquest punt és on vàrem decidir afegir un altre tecnologia: l'Entity Framework.

Finalment a la PAC 3, vàrem decidir que enlloc d'implementar només les 3 tecnologies que demanava el treball seleccionat, inclouríem també l'ADO.NET per a poder comparar les dues formes d'accés a dades: la tradicional que representa ADO.NET i la més moderna, que gràcies a l'ORM permet vincular dos móns tant diferents com el relacional i el conceptual.

Val a dir que també ens va ser prou útil començar el treball amb una tecnologia ja coneguda com és ADO.NET, per posteriorment un cop ja teníem creada i comunicant la BBDD, i la nostra aplicació ja prenia forma, anar augmentant progressivament la complexitat d'aquesta a mesura que vàrem anar entrant en les implementacions dels diferents frameworks.

1.8 Productes obtinguts

Els productes obtinguts durant aquest treball han estat els previstos en la planificació inicial:

- Document "Pla de treball".
- Document "Recerca, estudi i documentació" de les tecnologies seleccionades.
- Document "Anàlisi, disseny i implementació", que inclou també l'arquitectura del projecte i els manuals d'usuari i d'instal·lació i configuració.
- Generació del programari "Aplicació d'accés a dades", desenvolupat amb l'IDE Microsoft Visual Studio 2008.
- Generació de la BBDD "TFC" sota el SGBD Microsoft SQL Server 2008.
- Document "Memòria del projecte".
- Presentació virtual.

2 Estudi, recerca i documentació

2.1 Cronologia de tecnologies Microsoft d'accés a dades

ADO. Net és l'API actual d'accés a dades de la plataforma .NET, però anteriorment Microsoft va tenir altres biblioteques que proporcionaven aquesta funcionalitat. En farem un breu resum cronològic doncs no entra dins l'abast d'aquest TFC descriure un producte tan conegut, si no donar-ne només la introducció bàsica que ens permeti aprofundir tot seguit en la resta de tecnologies a estudi.

- 1992: apareix l'ODBC, sobre el qual també si van desenvolupar posteriorment DAO i RDO. Era una tecnologia complexa, lenta i no basada en el model de components COM.
- 1996: surt al mercat l'OLEDB, successora de l'ODBC que ja estava basada en COM. Per sobre s'hi va desenvolupar ADO per a substituir DAO i RDO.
- 2002: neix l'ADO.NET, la tecnologia d'accés a dades de .NET, que va heretar les millors característiques d'ADO i que a més proporcionava noves funcionalitats com ara poder treballar de forma desconnectada.

2.2 ADO.NET

ADO. Net és un conjunt de classes integrades dins el Microsoft .NET Framework, que proporciona accés a bases de dades relacionals i a d'altres tipus de dades per mig d'OLE DB i XML.

Una de les seves millors característiques és que permet treballar de forma desconnectada, encara que en té moltes altres, com ara que està fortament integrat amb XML, que és independent del llenguatge que fem servir, i que permet accedir a altres fonts de dades com fulls de càlcul, text, etc.

Veiem seguidament l'arquitectura d'ADO.NET , que es mostra a la imatge inferior:

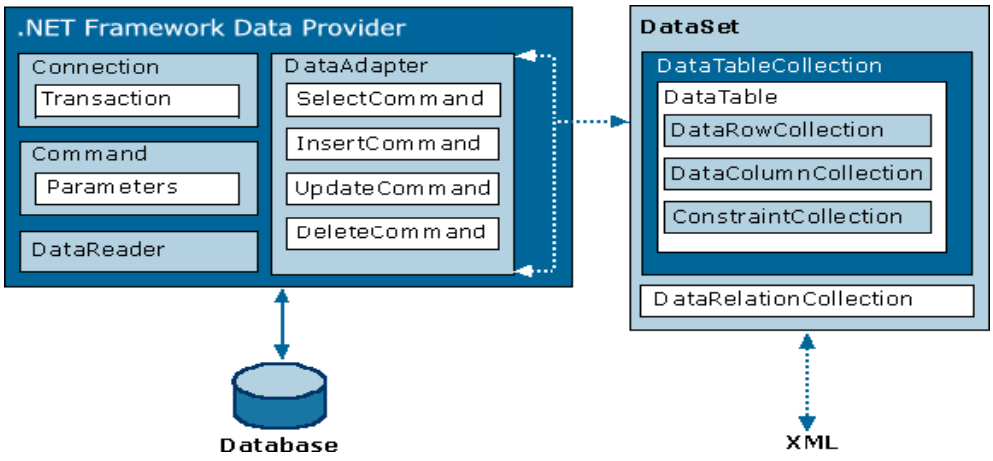


Figura 2. Arquitectura d'ADO.NET

Com podem veure, aquesta arquitectura es composa de dues parts primàries.

La primera, heretada directament d'ADO, és la Data Provider, que proporciona l'accés a la font de dades i que es composa de quatre classes principals: Connection, Command, DataReader (mode connectat) i DataAdapter (mode desconnectat) com veiem al reqüadre de l'esquerra de la imatge.

Els objectes Connection i Command necessiten d'un proveïdor específic en funció del SGBD al que ens vulguem connectar. Els principals són: SQL Client, Oracle Client, ODBC, OLEDB i Common, aquest últim usat quan no sabem encara amb quina base de dades haurem de connectar.

La classe Connection és l'encarregada de definir de quina manera i a quina BBDD ens connectarem. Command, per la seva banda, és l'encarregada d'accedir a les dades i de transportar-les fins a ser agafades per als objectes DataReader si treballem amb connexió o DataAdapter en mode desconnexió.

La segona part primària, que veiem al reqüadre dret de la imatge superior, és l'objecte DataSet, l'estrella del llançament d'ADO.NET, un grup de classes que descriuen un model de base de dades relacional en memòria, sigui la BBDD sencera o un subconjunt d'aquesta. Pot contenir totes les taules i les seves relacions, com veurem més endavant quan tractem del mode desconnectat

A la figura inferior podem veure gràficament el model de transport de dades.

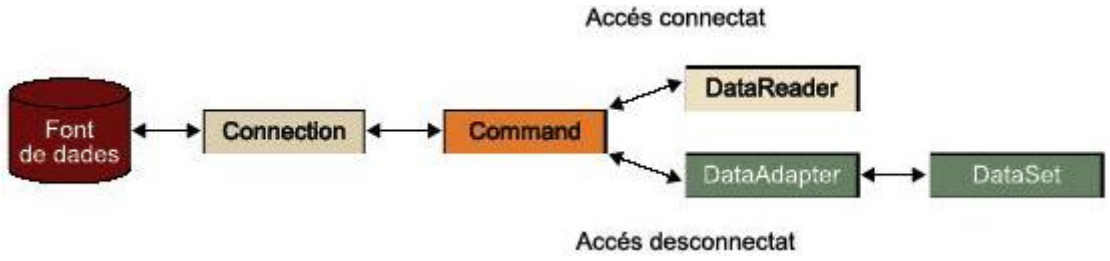


Figura 3. Modes d'accés d'ADO.NET

No afegirem més en aquesta breu introducció a ADO.NET, ja que és un producte prou conegut i que només ha estat incorporat en aquest treball com a nexe entre el model tradicional d'accés a dades i el que proporcionen les modernes tecnologies que segueixen el patró ORM, on si que serem més extensos ja que han format una part molt important d'aquest treball, tant en la part de recerca i documentació con en la d'implementació.

2.3 ORM

2.3.1 Introducció

El marc de Mapeig Objecte Relacional (ORM en endavant) és una tècnica de programació per a solucionar el problema de la conversió de dades entre el format de BBDD relacional i el format conceptual emprat a la orientació a objectes.

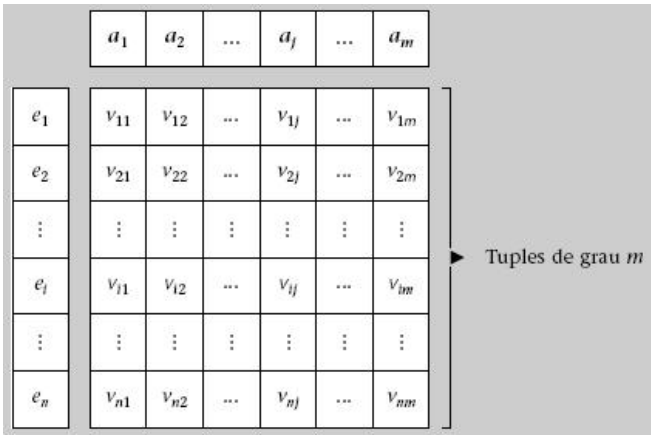


Figura 4. Representació tabular usada en el model relacional

Per una banda, el model relacional és un model de dades basat en la lògica de predicats i en la teoria de conjunts. Conceptualment es basa fonamentalment en l'ús de relacions, pensant en cadascuna d'aquestes com si fossin una taula que té registres (cada fila seria una tupla) i columnes. Per tant una base de dades relacional és un conjunt de taules estructurades en files i columnes, vinculades per un camp en comú: la clau primària. A aquest tipus de construir les BBDD se l'anomena model relacional, el qual usa una representació tabular.

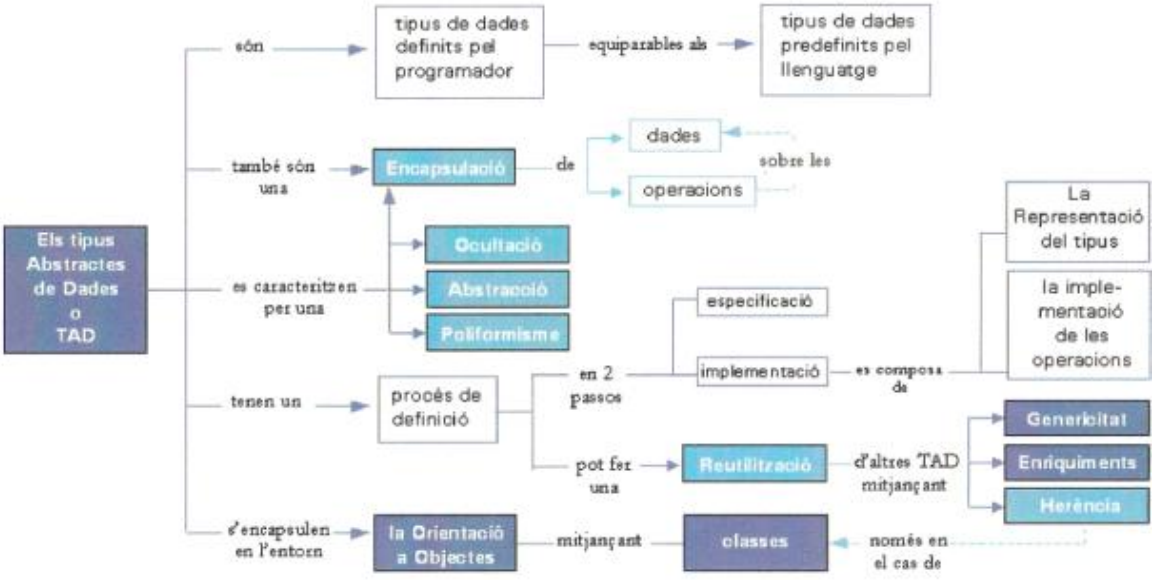


Figura 5. Tipus abstractes de dades usats a l'orientació a objectes

Per altra banda, en el paradigma de la programació orientada a objectes (OO) es manipulen tipus abstractes de dades (TAD) que són quasi sempre valors no escalars i que tenen una representació jeràrquica.

El problema el tenim en que els SGBD SQL només poden manipular valors escalars primaris con enters i cadenes, i és el programador el que hauria de convertir els valors dels objectes en grups de valors simples per a poder-los emmagatzemar a la BBDD, per posteriorment tornar a fer l'operació inversa quan s'haguessin de recuperar aquestes dades i tornar-les a convertir en el model conceptual d'objectes. El mateix tipus de dades que es

poden emmagatzemar en un sol objecte, en el model relacional poden requerir ser emmagatzemats per mig de diverses taules.

Com es veu doncs, no és trivial traduir els objectes a tipus de dades que puguin ser fàcilment emmagatzemades en una base de dades no orientada a objectes mantenint les propietats d'aquests i les seves relacions, per això es van començar a desenvolupar diferents patrons ORM.

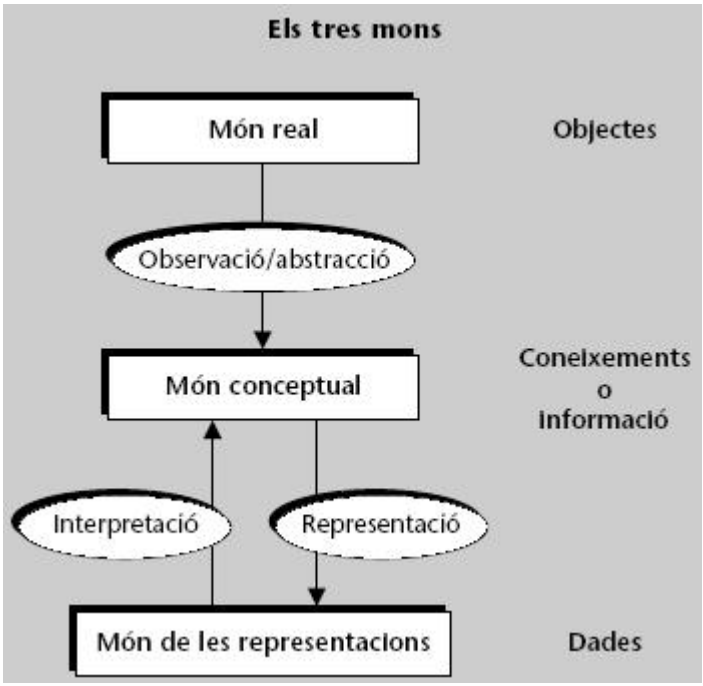


Figura 6. Els tres móns: el real, el conceptual i el de representació de dades

Val a dir que el model relacional és l'emprat per les BBDD més usades actualment, com ara: Oracle, SQL Server, MySQL, etc., però no és l'únic. Les bases de dades com Caché no necessiten el mapeig ORM ja que l'accés de l'SQL als valors no escalars ja ha estat construït. Aquesta solució permet dissenyar qualsevol combinació de programació OO i emmagatzematge estructurat en taules, a la mateixa base de dades enlloc de dependre d'eines externes.

També existeixen els SGBD OO, que ja estan expressament dissenyats per a treballar amb objectes i les dades poden ser persistides en el seu format original. Aquests sistemes tenen però l'inconvenient principal de que no permeten crear sentències SQL, només processar-les de forma limitada. De moment doncs, cap de les dues solucions alternatives ha assolit encara un alt nivell d'acceptació.

2.3.2 Cronologia dels sistemes ORM

Al llarg dels anys s'han anat desenvolupant diversos sistemes de mapeig objecte-relacional. Un dels millors va ésser NeXT's Enterprise Objects Framework (EOF) però estava massa lligat al programari OpenStep i no va tenir èxit.

Més tard van aparèixer tecnologies com RDF i SPARQL i el concepte de "triplestore". El primer és una serialització del concepte objecte-subjecte-predicat. El segon és un llenguatge similar a l'SQL, i el "triplestore" és una descripció general d'una BBDD que treballa amb un tercer component.

Més recentment, dins el mon Java ha començat a evolucionar un concepte similar conegut com Java Data Objects (JDO) que a diferència d'EOF si que és un estàndard amb moltes implementacions conegudes. També ha sortit l'especificació 3.0 d'Enterprise Java Beans (EJB) encara que aquest altre està encara en desenvolupament.

Un altre exemple més conegut és Hibernate, el framework ORM més usat en Java, que facilita el mapeig d'atributs mitjançant arxius declaratius (XML) o anotacions als "beans" de les entitats que permeten establir aquestes relacions. Aquesta mateixa tecnologia disposa de la versió NHibernate, que és la conversió de Java a C# per a la seva integració a la plataforma .NET, la qual estudiarem a l'apartat 4.

Veurem també el model proposat per ADO.NET: l'Entity Framework, objecte que té una clau representant la clau primària d'una entitat lògica relacional i que usant l'Entity Data Model permet que les dades siguin tractades com entitats, independentment de les seves representacions del "datastore" subjacent.

2.3.3 Mapeig

Un bon ORM ha de permetre:

- El **pas del model conceptual al relacional**, mapejant les classes a taules on les propietats són les columnes i les files els objectes (o instàncies de la classe).
- La **persistència dels objectes**, encarregant-se de generar les sentencies SQL apropiades.
- El **pas del model relacional al model conceptual**, permetent la recuperació dels objectes persistits.

Un objecte es composa bàsicament de propietats i mètodes, on les propietats són les parts estàtiques que han de ser persistides i poden ser simples o complexes. Els simples són els tipus de dades natius (booleà, enter, caràcter, etc.) i els complexes són els creats per l'usuari (tuples o altres objectes).

Cada columna hauria de respectar el mateix tipus de dades que la propietat de l'objecte, encara que de vegades, per optimització, es realitzen alguns ajustaments a la taula com ara canviar cadenes o caràcters per als seus equivalents numèrics, aconseguint així un rendiment molt més alt.

Per a garantir que no es perdi la identitat de l'objecte, cal afegir una columna numèrica identificativa d'aquest (OID) la qual sempre representarà la clau primària. Aquest OID s'usarà també per al mapeig de relacions, agregant-se com una columna més (clau forània) a la taula amb la que es vol establir la relació.

Per a mapejar i persistir les relacions (d'associació, herència o agregació) entre classes, s'usen les transaccions, ja que els canvis poden afectar a diverses taules. Una regla general és respectar el tipus de multiplicitat, així una relació 1:1 al model d'objectes, haurà de correspondre a la mateixa relació al model relacional.

Alhora, les associacions es divideixen en funció de la multiplicitat i la navegabilitat. Segons la primera, poden haver-hi associacions 1:1, 1:N o M:N, i segons la segona poden ser unidireccionals o bidireccionals, però en el model relacional totes les associacions són bidireccionals, un altre factor de la incongruència de models.

Anem a introduir ara com es fa el mapeig. Per a mapejar la jerarquia a una taula, els atributs de totes les classes de l'arbre d'herència es mapegen a una sola taula, a la qual s'afegeixen dues columnes, una amb l'OID ja comentat i l'altra amb el tipus de classe que és cada registre.

Posem un exemple: una jerarquia amb la superclasse abstracta Persona i dues subclasses Client i Empleat, quedarien en una sola taula anomenada Persona. A la taula s'afegiria la columna TipusPersona, on C podria ser un client, E un empleat i D un client i empleat alhora.

Per a mapejar les classes, es mapegen cadascuna d'elles (incloses les abstractes) a una taula per classe, a la qual han d'haver-hi tan els atributs heretats com els propis de la subclasse. Una classe abstracta es mapejada doncs a cadascuna de les seves implementacions. S'afegeixen també columnes per al control de la concurrència o versió a qualsevol de les taules. Les claus primàries de totes les taules heretades han de ser les mateixes que la taula base i alhora seran les claus foranies envers aquesta.

Alhora de persistir les dades, el framework ORM que usem ha de resoldre diversos problemes, com ara: la gestió de les transaccions i de la memòria cau, la càrrega retardada de l'objecte en memòria, la referència circular i l'assignació dels codis OID.

2.3.4 Altres aspectes

Altres aspectes a valorar alhora de triar un framework que implementi el patró ORM, són:

- **La referència circular**. Aquest concepte es refereix a si el framework es capaç de detectar quin és l'objecte sol·licitant. sense haver de fer el "roundtrip" a la BBDD.
- **La informació oculta**. Coneguda amb el seu nom en anglès "Shadow Information" es refereix a que igual que l'OID, hi ha altres columnes que no necessiten ser mapejades a cap propietat de l'objecte. Tenen informació oculta del model d'objectes però que és necessària al model relacional, com ara: els mecanismes de gestió de la concurrència, la versió de l'objecte, etc.
- **Llenguatge de consulta OQL (Object Query Languaje)**. La capacitat de mitjançant aquest llenguatge, poder adquirir diversos objectes en una sola consulta. NHibernate amb HQL és un dels millors.
- **Actualització en cascada**. La possibilitat de que les modificacions que fem sobre un objecte afectin a tots els que hi té relacionats.
- **Operacions en massa**. La prestació de poder fer operacions massives.

2.3.5 Principals avantatges

El fet que al programar no calgui pensar en la persistència de les dades (taules, columnes, relacions, etc.) permet als arquitectes i programadors treballar en un nivell molt més alt d'abstracció, i així poden crear i mantenir aplicacions OO amb menys codi que les aplicacions tradicionals. L'objectiu bàsic d'implementar el patró ORM és doncs, reduir la quantitat de codi i manteniment que es necessita per a les aplicacions orientades a dades. Usar-lo té avantatges com ara:

- Treballar directament sobre el model conceptual OO, amb els seus membres complexos i les seves relacions.
- Les aplicacions no estan sotmeses a les normes rígides de codificació d'un motor de dades o d'un esquema d'emmagatzematge.
- Les assignacions entre aquest darrer i el model conceptual poden canviar sense tenir que canviar el codi de l'aplicació.
- Els programadors poden treballar amb un model d'objecte coherent que es pot assignar a diversos esquemes d'emmagatzematge implementats en SGBD diferents.
- Es poden assignar diversos models conceptuals a un únic esquema de persistència.

Seguidament estudiarem els principals frameworks que implementen el patró ORM.

2.4 Entity Framework

2.4.1 Introducció

L'ADO.NET Entity Framework (d'ara endavant EF) és un conjunt d'APIs d'accés a dades per al Microsoft .NET Framework. La primera versió (amb la que hem topat en aquest TFC) venia inclosa al .NET Framework 3.5 Service Pack 1 després del llançament del Visual Studio 2008. Posteriorment s'ha tret una segona versió que ve amb el Framework 4.0 i el Visual Studio 2010.

L'EF és un objecte que té una clau primària d'una entitat lògica relacional i que permet als programadors crear aplicacions d'accés a dades programant sobre el model de l'aplicació conceptual enlloc de fer-ho directament sobre el model relacional, és a dir, implementa el patró ORM.

2.4.2 L'Entity Data Model (EDM)

Abans d'entrar més a fons en l'EF, cal donar una breu introducció a l'EDM. Aquest és un conjunt de conceptes que descriuen l'estructura de les dades independentment del format en que estiguin emmagatzemats. Es basa en el patró ORM però alhora incorpora noves funcions i amplia els seus usos tradicionals.

EDM soluciona la problemàtica de tenir dades guardades en diversos formats. Al dissenyar una aplicació orientada a dades cal escriure un codi eficaç i fàcil de mantenir sense sacrificar l'eficàcia d'accés a dades, d'emmagatzematge i d'escalabilitat. Tan el model relacional com

el conceptual tenen els seus avantatges i inconvenients en aquests aspectes, i encara que es podria trobar un equilibri entre els dos móns, apareixen nous inconvenients al moure les dades entre l'un i l'altre format.

EDM soluciona aquests inconvenients descrivint l'estructura de les dades en forma d'entitats i relacions que són independents de qualsevol esquema d'emmagatzematge. Això fa que el format de les dades sigui irrellevant alhora de dissenyar i desenvolupar les aplicacions. Com que les entitats i les seves relacions descriuen les estructures de dades tal i com s'usen a les classes de negoci, poden evolucionar igual que l'aplicació.

Un model conceptual és una representació específica de l'estructura de les dades en forma d'entitats i relacions, i normalment es defineix mitjançant un llenguatge específic de domini (DSL) que implementa els conceptes de l'EDM. El llenguatge de definició d'esquemes conceptuals (CSDL) és un exemple d'aquest tipus de llenguatge específic de domini. Les entitats i relacions descrites en un model conceptual es poden considerar com abstraccions d'objectes i associacions en una aplicació, permetent als desenvolupadors centrar-se en el model conceptual sense que tenir que preocupar-se per l'esquema d'emmagatzematge.

Per a consultar l'EDM existeix l'Entity SQL (ESQL), un llenguatge similar a l'SQL. Aquestes consultes són convertides internament a un "Canònical Query Tree", que a la seva vegada es convertit a una consulta SQL, la qual si que ja és comprensible per a la base de dades relacional. Les entitats poden usar les seves relacions i les seves modificacions ser enviades de tornada a la BBDD.

2.4.3 Eines de l'EDM

Les eines de l'EDM generen classes de dades extensibles segons el model conceptual. Es tracta de classes parcials que es poden estendre com a membres addicionals que el programador afegeix. De forma predeterminada, les classes que es generen per a un model conceptual determinat deriven de les classes base que proporcionen serveis per a materialitzar les entitats com a objectes i per a realitzar un seguiment dels canvis i guardar-los. Aquestes classes es poden usar per a treballar amb les entitats i relacions com si fossin objectes relacionats mitjançant associacions. També es possible personalitzar les classes que es generen per a un model conceptual.

Aquestes eines poden generar una classe derivada d'ObjectContext, que representa el contenidor d'entitats definit al model conceptual. Aquest context de l'objecte proporciona els mitjans per a realitzar el seguiment dels canvis i administrar les entitats, la simultaneïtat i les relacions. Aquesta classe també té un mètode SaveChanges() que escriu les insercions, actualitzacions i eliminacions a l'origen de dades. Com passa amb les consultes, aquestes modificacions són realitzades o bé per les comandes que el sistema genera automàticament o bé per als procediments emmagatzemats que el programador especifica.

Com a novetat a la versió 4 del .NET Framework, s'inclou el generador d'EDM (EdmGen.exe que també hem usat en aquesta versió) utilitat de línia de comandes que permet generar un model conceptual simple. Es connecta amb un origen de dades i genera arxius del model i d'assignació basats en una assignació unívoca entre les entitats i les taules. També usa un arxiu de model conceptual (.csdl) per a generar un arxiu de nivell d'objecte que conté classes que representen tipus d'entitat i Object Context.

Visual Studio 2010 inclou una variada compatibilitat amb les eines que permeten generar i mantenir arxius de model i d'assignació a una aplicació VS. L'EDM Designer permet crear escenaris d'assignació avançats i entitats de divisió que s'assignen a diverses taules.

Val la pena comentar també que aquestes eines permeten generar automàticament el contingut CSDL, SSDL i MSL que introduirem tot seguit. L'assistent per a l'EDM genera la informació del model, l'assignació, i les classes de dades, a partir d'una BBDD existent.

2.4.4 Els models de l'EDM

Alhora de dissenyar una l'aplicació, un enfocament de disseny habitual consisteix en dividir-la en tres models: el de domini, el lògic i el físic. El primer defineix les entitats i relacions del sistema a modelitzar, és la conceptualització del món real. El segon, el model lògic de la base de dades relacional, normalitza les entitats i relacions en taules amb restriccions de claus externes. El darrer, el físic, avarca les capacitats d'un motor de dades determinat especificant els detalls d'emmagatzematge en forma de particions i índexs.

EF dona vida als models conceptuals permetent als programadors treballar sobre el domini de negoci alhora que es basen en EF per a traduir les operacions necessàries a comandes especifiques de l'origen de dades. Això allibera a les aplicacions d'implementar codis rígids dependents del proveïdor de dades. L'EDM consta de tres parts: el model conceptual, el d'emmagatzematge i les assignacions entre ambdós, que s'expressen en esquemes basats en XML i es defineixen en arxius que tenen extensions de nom corresponents.

- El llenguatge de definició d'esquemes conceptuals (conegut amb l'acrònim CSDL) defineix el model conceptual, declarant i definint entitats, associacions, herència, etc. És una implementació d'EF de l'EDM. La seva extensió d'arxiu és .csdl.
- El llenguatge de definició d'esquemes d'emmagatzematge (SSDL) defineix aquest model, també anomenat lògic. La seva extensió és .ssdl.
- El llenguatge d'especificació d'assignacions (MSL) defineix els mapejos entre els models conceptual i d'emmagatzematge. En altres paraules, mapeja les entitats del fitxer CSDL a les taules descrites al fitxer SSDL. La seva extensió és .msl.

El model d'emmagatzematge i les assignacions poden canviar sense que això representi canvis en el model conceptual, les classes de dades o el codi de l'aplicació. Com els models d'emmagatzematge són específics del proveïdor, es pot treballar amb un model conceptual coherent per mig de diversos orígens de dades. EF usa aquests models i arxius d'assignació per a transformar les operacions bàsiques SQL en operacions equivalents a l'origen de dades.

La programació OO suposa un desafiament al interactuar amb bases de dades relacionals. Tot i que l'organització de classes acostuma a reflectir l'organització de les taules de la BBDD, aquest ajustament no és perfecte. Com hem introduït a l'estudiar el patró ORM, sovint diverses taules corresponen a una sola classe i les relacions entre aquestes es representen amb freqüència de forma diferent a les relacions entre taules. EF proporciona la flexibilitat per a representar relacions d'aquesta forma o per a modelar millor les relacions tal i com es representen a la base de dades.

Tradicionalment, les solucions ORM existents han intentat cobrir aquest forat, anomenat desigualtat d'impedància, assignant únicament classes i propietats OO a les taules i columnes relacionals. EF, enlloc de seguir aquest enfocament tradicional assigna a les taules, columnes i restriccions claus foranes dels models lògics a les entitats i relacions dels models conceptuals. Això permet una major flexibilitat al definir els objectes i optimitzar el model lògic.

2.4.5 Arquitectura i accés a dades

Com a diferenciació de la resta de frameworks ORM, l'EF tracta fonamentalment de permetre que les aplicacions obtinguin accés i puguin canviar les dades que estan representades com a entitats i relacions al model conceptual. EF usa la informació dels arxius del model i d'assignació per a traduir les consultes d'objecte amb els tipus d'entitat que es representen al model conceptual, en consultes especifiques de l'origen de dades. Els resultats de la consulta es materialitzen en objectes que l'EF administra. Existeixen tres formes de consultar un model conceptual i retornar objectes:

- **LINQ to Entities**. Proporciona compatibilitat amb LINQ per a consultar els tipus d'entitat que es defineixen en un model conceptual (a l'EDM).
- **Entity SQL** (ESQL). Un dialecte de Transact-SQL independent de l'emmagatzematge, dissenyat per a treballar amb entitats definides a l'EDM. Suporta herència i associacions. L'ESQL s'usa tan amb consultes d'objecte com en consultes que s'executen amb el proveïdor Entity Client.
- **Mètodes del generador de consultes**. Aquests mètodes permeten construir consultes d' ESQL usant mètodes de consulta de l'estil de LINQ.

Per sota d'aquesta capa apareix l'Object Services, un component que permet realitzar les quatre operacions bàsiques de l'SQL: consultar, inserir, actualitzar i esborrar, expressades com objectes CLR amb tipus que són instàncies de tipus d'entitat.

L'Object Services alhora comunica amb el proveïdor de dades Entity Client, el qual permet accedir a les dades descrites a l'EDM. Administra les connexions, tradueix les consultes d'entitat en consultes específiques de l'origen de dades i retorna un lector de dades que l'EF usa per a materialitzar les dades de l'entitat als objectes. És similar a SQLClient, OracleClient, etc. Proporciona components com EntityCommand, EntityConnection i EntityTransaction.

L'Entity Client a la seva vegada interactúa amb el model conceptual que proporciona l'EDM. Dins aquest, hi ha tres arxius (CSDL, SSDL i MSL) ja descrits anteriorment a l'apartat 3.4. L'EDM és l'encarregat de parlar amb els proveïdors de dades d'ADO.NET i aquests són els que proporcionen l'accés a la base de dades.

Tot seguit mostrem dues versions de l'arquitectura d'EF per a l'accés a dades:

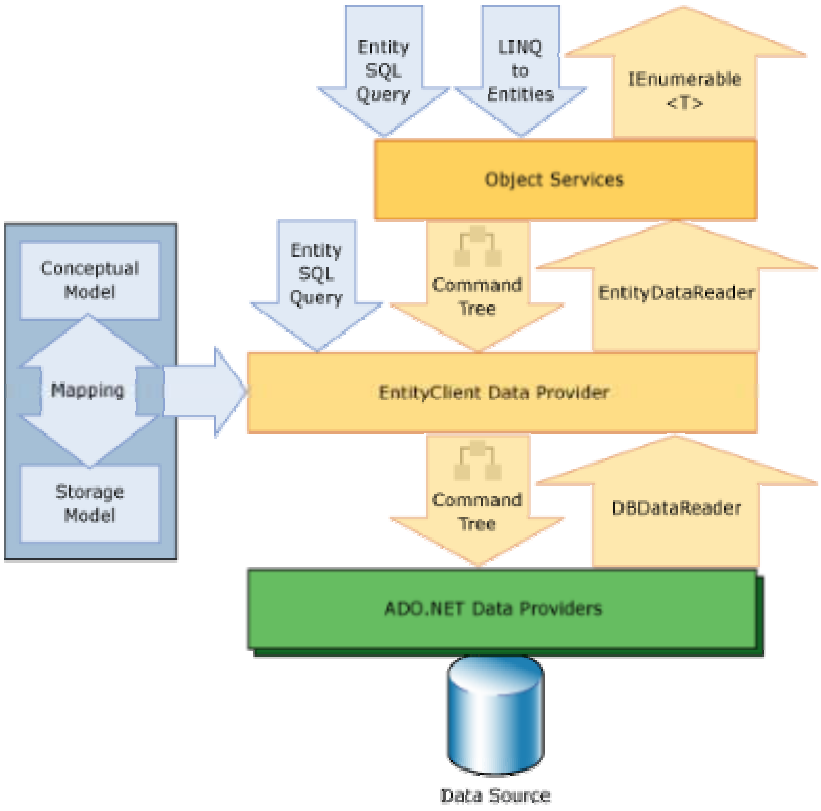


Figura 7. Arquitectura d'EF (font: msdn)

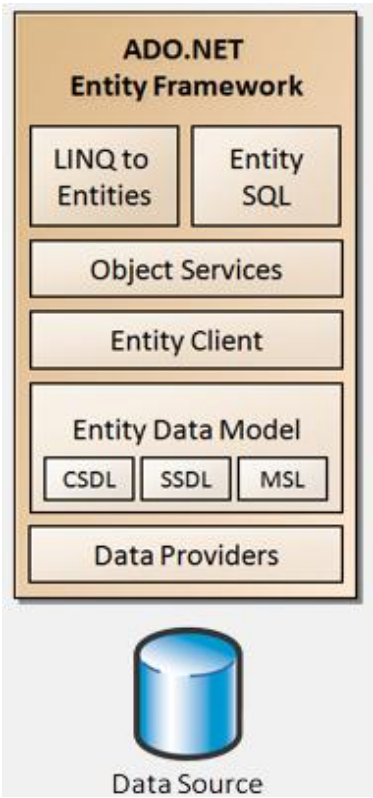


Figura 8. Arquitectura d'EF (font: scip.be)

Podem observar que ambdues difereixen lleugerament alhora d'explicar la interactuacció entre l'Entity Client i l'EDM, bàsicament perquè la figura 19 fa un major abstracció del model conceptual. Tot i que aquesta arquitectura és més correcta que no pas la de la figura 20, hem afegit aquesta darrera només per a poder visualitzar en major detall els components de l'EDM. A l'apartat següent explicarem millor com és l'Entity Client qui comunica amb el proveïdor d'ADO.NET.

2.4.6 Cadenes de connexió

Una cadena de connexió conté informació d'inicialització que es transfereix com un paràmetre des d'un proveïdor de dades a un origen de dades. La sintaxi depèn del proveïdor i la cadena s'analitza mentre s'intenta obrir una connexió. Les cadenes que usa Entity Framework contenen la informació que s'usa per a connectar amb el proveïdor de dades ADO.NET subjacent que l'EF admet. També contenen informació sobre els arxius del model i d'assignació necessaris. El proveïdor d'Entity Client usa la cadena de connexió a l'obtenir accés a les metadades del model i d'assignació, i al connectar amb l'origen de dades.

Les eines de l'EDM generen una cadena de connexió que s'emmagatzemen a l'arxiu de configuració de l'aplicació. ObjectContext recupera aquesta informació de connexió automàticament al crear consultes d'objectes. Es pot tenir accés a l'element EntityConnection que usa una instància d'ObjectContext des de la propietat Connection.

El format d'una cadena de connexió és una llista de parells de paràmetres de clau i valor delimitats per punt i coma: "keyword1=value; keyword2=value;". El signe igual (=) associa cada paraula clau al seu valor. Els noms vàlids com a paraula clau a la propietat ConnectionString són els següents: provider, provider connection string, metadata i name. Metadata contindria una llista d'ubicacions a les que el proveïdor EntityClient buscaria els arxius del model i d'assignació.

2.4.7 Proveïdors de dades, consultes i transaccions

El proveïdor d'Entity Client és un proveïdor de dades que usen les aplicacions d'Entity Framework per a tenir accés a les dades descrites a un model conceptual. Entity Client usa altres proveïdors de dades .NET Framework per a tenir accés a l'origen de dades, com per exemple el .NET Framework per a SQL Server (SqlClient).

Entity Framework es basa en proveïdors de dades ADO.NET específics de l'emmagatzematge, proporcionant un objecte EntityConnection a un proveïdor de dades subjacent i una base de dades relacional.

El proveïdor Entity Client estén el model de proveïdor d'ADO.NET, tenint accés a les dades en el que respecta a les entitats conceptuals i relacions. Executa consultes que usen Entity SQL, el qual proporciona el llenguatge de consultes subjacent que permet a Entity Client comunicar-se amb la base de dades.

El llenguatge d'ESQL és un dialecte independent de l'emmagatzemament d'SQL que treballa directament amb esquemes d'entitats conceptuals i admet conceptes d'EDM, com l'herència i les relacions. La classe EntityCommand s'usa per a executar una comanda d'ESQL amb un model de l'entitat.

Quan s'executen consultes s'analitza i es converteix en un arbre de comandes canònic. Tot el processament següent es realitza a l'arbre de comandes, que és el mitja de comunicació entre l'espai de noms System.Data.EntityClient i el proveïdor de dades .NET Framework subjacent, com System.Data.SqlClient.

EntityDataReader mostra els resultats d'executar EntityCommand en un model conceptual. Per a executar la comanda que retorna l'EntityDataReader, cal cridar a ExecuteReader. EntityDataReader, que implementa iExtendedDataRecords per a descriure resultats estructurats enriquits.

Introduïm breument el tema de les transaccions. Una transacció és un servei proporcionat per les bases de dades o qualsevol altra administrador de recursos, que es pot usar per a garantir que una sèrie d'accions individuals es produeixin de forma automàtica. A l'Entity Framework hi ha dues maneres d'usar les transaccions: automàtica i explícita. Les automàtiques usen l'espai de noms System.Transactions i les explícites la classe Entity Transaction.

Seguidament donarem una breu introducció al proveïdor de dades .NET Framework per a SQL Server (SqlClient), que permet a Entity Framework treballar sobre Microsoft SQL Server.

Provider és un atribut de l'element Schema de l'SSDL. Per a usar SqlClient cal assignar la cadena "System.Data.SqlClient" a l'atribut Provider de l'element Schema. ProviderManifestToken és un altre atribut necessari del mateix element en SSDL. Aquest token s'usa per a carregar el manifest del proveïdor en escenaris sense connexió.

SqlClient es pot usar com a proveïdor de dades per a diferents versions d'SQL Server, encara que cadascuna té una capacitat diferent, per exemple la versió 2000 no inclou alguns tipus que es van incloure a la 2005.

Tots els proveïdors han d'especificar un espai de noms. Aquesta propietat indica a l'EF quin prefix usa el proveïdor per a estructures concretes, com els tipus i les funcions. L'espai de noms per SqlClient és SqlServer.

Finalment només comentar que el .NET Framework inclou proveïdors per accedir directament a Microsoft SQL Server i per a accedir indirectament a altres BBDD amb drivers ODBC i OLEDB. Alguns dels proveïdors disponibles són:

- VistaDB,
- Devart,
- OpenLink,
- DataDirect,
- IBM,
- Sybase SQL Anywhere,
- Phoenix,
- Synergex,
- Firebird,
- Npgsql.

2.4.8 Serialització amb EF

Repassem molt breument el concepte de serialització. Consisteix en un procés de codificació d'un objecte a un mitjà d'emmagatzematge (com un arxiu o un buffer de memòria) amb la finalitat de poder-lo transmetre com una sèrie de bytes o en un format més llegible com XML. La serialització és molt usada per a transportar objectes per una xarxa, per a fer-los persistents, o per a distribuir-los a diverses aplicacions o localitzacions.

Als tipus d'entitat que genera l'eina de generació de l'EDM (EdmGen.exe) i l'assistent per EDM, se'ls apliquen SerializableAttribute i DataContractAtribut. Això permet serialitzar els objectes mitjançant les serialitzacions:

- **binària**,
- **XML**,
- i **de contracte de dades** de Windows Communication Foundation (WCF).

Durant la serialització i deserialització d'entitats, s'apliquen les següents consideracions:

- Al serialitzar les entitats, l'usuari hauria de considerar la possibilitat de deshabilitar la càrrega diferida, sinó s'usarà aquesta i el gràfic d'objectes serialitzat podria incloure més dades de les esperades.
- Al serialitzar un objecte també es serialitza l'objecte EntityKey.
- Quan s'usa la serialització binària i la de contracte de dades de WCF, l'objecte que s'està serialitzant té objectes relacionats en el gràfic d'objectes, que també es serialitzen. La serialització XML no serialitza objectes relacionats.
- Només es serialitzen les propietats dels objectes i la informació de relació. No es serialitza la informació d'estat dels objectes, que es guarda al context d'aquests. A partir de la versió 4 de .NET Framework, les entitats amb seguiment propi poden contenir la seva pròpia lògica de seguiment de canvis.
- Un cop deserialitzat, l'objecte es troba a l'estat "Detached".

Com els tipus d'entitat admeten la serialització binària, els objectes es poden guardar en l'estat de vista d'un aplicació ASP.NET durant una operació "postback". Quan es requereix, l'objecte i els seus objectes relacionats es recuperen de l'estat de vista i s'adjunten a un context d'objectes persistents.

2.5 NHibernate

2.5.1 Introducció a Hibernate

NHibernate és una eina hereva d'Hibernate, així que abans de res farem una breu introducció a aquesta tecnologia.

Hibernate és un framework de software lliure, distribuït sota la llicència GNU LGPL, que implementa el patró ORM per a la plataforma Java, facilitant així el mapeig d'atributs entre una BBDD relacional i el model d'objectes d'una aplicació, mitjançant arxius XML o anotacions al "beans" de les entitats que permeten establir aquestes relacions.

Per a assolir la implementació del patró es permet al desenvolupador descriure com és el model de dades, quines relacions existeixen, i quina forma tenen. Amb aquesta informació, Hibernate permet a l'aplicació manipular les dades de la base operant sobre els objectes, amb totes les característiques de la programació orientada a objectes.

Hibernate convertirà les dades entre els tipus usats per Java i els definits per SQL, generarà les sentencies SQL, i alliberarà al programador de la manipulació manual de les dades que resulta de la introducció d'aquestes sentències, mantenint la portabilitat entre tots els motors de BBDD, amb un lleuger increment al temps d'execució.

És una eina dissenyada per a ser flexible en quant a l'esquema de taules utilitzat, per a poder adaptar-se al seu ús sobre una base de dades ja existent. També té la possibilitat de poder crear la pròpia BBDD a partir de la informació disponible. A més, ofereix un llenguatge de consulta de dades anomenat Hibernate Query Language (HQL) i una API per a construir les consultes programàticament, coneguda com "criteria".

Hibernate per a Java pot ser usat en aplicacions Java independents o en aplicacions JavaEE (arquitectura client - servidor) mitjançant el component Hibernate Annotations, que implementa l'estàndard Java Persistence Api (JPA) que és part d'aquesta plataforma.

El naixement d'aquesta tecnologia va ser una iniciativa d'un grup de desenvolupadors dispersos arreu del món, conduïts per Gavin King. Més endavant, JBoss Inc. (empresa comprada per Red Hat) va contractar als principals programadors en aquesta tecnologia i va treballar amb ells per a oferir suport al projecte.

La branca actual de desenvolupament és la 3.x, que incorpora noves característiques, com a una nova arquitectura "Interceptor/Callback", filtres definits per l'usuari, i opcionalment l'ús d'anotacions per a definir la correspondència enlloc o alhora dels arxius XML. Hibernate 3.0 també guarda la proximitat amb l'especificació Enterprise Java Beans (EJB) 3.0 i actua com eix central de la implementació d'aquesta a JBoss.

2.5.2 Introducció a NHibernate

NHibernate (en endavant NH) és la conversió d'Hibernate del llenguatge Java a C# per a la seva integració a la plataforma .NET de Microsoft. A l'igual que moltes altres eines lliures per a aquesta plataforma, NH també funciona en el Projecte Mono, projecte de codi obert per a crear un grup d'eines lliures, basades en GNU/Linux i compatibles amb .NET.

Si usem NH per a l'accés a dades, el desenvolupador s'assegura de que la seva aplicació és agnòstica en quant al motor de BBDD a usar en execució, doncs aquesta tecnologia suporta els més habituals del mercat. Només cal canviar una sola línia al fitxer de configuració per que puguem usar una base de dades diferent.

Com la seva tecnologia mare, NH també és software lliure, distribuït sota els termes de la Llicència Pública General Menor (LGPL) de GNU.

Així doncs, podem concloure aquesta introducció dient que NH és un framework d'ORM de codi lliure que soluciona de forma automàtica la persistència d'objectes de domini a la plataforma .NET.

2.5.3 Principals característiques

Els principals avantatges que té aquesta tecnologia són:

- Model de programació OO, que suporta herència, polimorfisme, etc.
- És nadiu .NET, la seva API usa convencions i idiomes .NET.
- El codi obert.
- Suporta "DataBinding".
- Pot acceptar consultes directes SQL.
- Es pot integrar amb altres frameworks com MVC o metaframeworks com Spring.
- Suporta: Read i Write Batching, relacions, agrupament, claus primàries compostes, associacions M:N i 1:M, persistència de propietats, treballar desconnectats de la BBDD, Webservices i tipus nuls.

- No cal generar codi precompilat.

I els principals inconvenients són:

- Generació del model complexa i mitjançant XML.
- No suporta la càrrega no transaccional "lazy" de relacions.
- No suporta "Aggregate Mappings – Single". 1:M a la BBDD.
- No suporta consultes transparents a múltiples recursos de data.

2.5.4 Comparació amb l'Entity Framework

Si fem una comparativa entre l'NH i l'EF estudiat al capítol anterior, podríem destacar-ne els següents aspectes a favor d'NH:

- Les eines anomenades Write/Read Batching, que permeten respectivament: guardar un bloc d'entitats, i l'execució de diverses consultes, en un sol viatge a la BBDD enlloc de realitzar-les una a una.
- Les col·leccions amb lazy="extra", utilitat intel·ligent que permet filar mes prim jugant amb la càrrega vaga i les càrregues de col·leccions per lots, que permet fer un "count" de les col·leccions sense necessitat de carregar totes les entitats de la base de dades i fer un item.Collection.Contains() que enlloc de carregar la col·lecció farà una simple consulta a la BBDD per veure si conté l'element sense necessitat de carregar la resta.
- La col·lecció "Filters & Paged Collections", que permet definir filtres a nivell de col·lecció o paginació de les mateixes a nivell de mapeig de les nostres entitats.
- El segon nivell de memòria cau, compartida entre totes les nostres sessions d'un mateix "Session Factory", fet que agilitza força la BBDD i que en web és molt senzill d'implementar sense patir masses riscos de concurrència.
- Tweaking (afinació), al ser una eina de codi obert es permet afinar més que amb l'EF.
- Integració i extensibilitat, que permeten ampliar-ne la funcionalitat amb altres paquets com: NH Search, NH Validator, NH Shards, NConstruct Lite, etc.
- És una tecnologia no tan novell com l'EF, que és encara massa jove.

Per altra banda, l'EF té a favor:

- Una millor integració amb LINQ i la resta de programaris de Microsoft, doncs no deixa de ser un producte més d'aquesta casa, cosa que es nota molt.
- Les eines gràfiques de que disposa, que eviten que el desenvolupador s'hagi d'estudiar en detall com funciona internament.
- No cal conèixer en detall l'XML.

2.5.5 Arquitectura i accés a dades

Una primera vista de l'arquitectura, en un nivell d'abstracció molt elevat, seria aquesta:

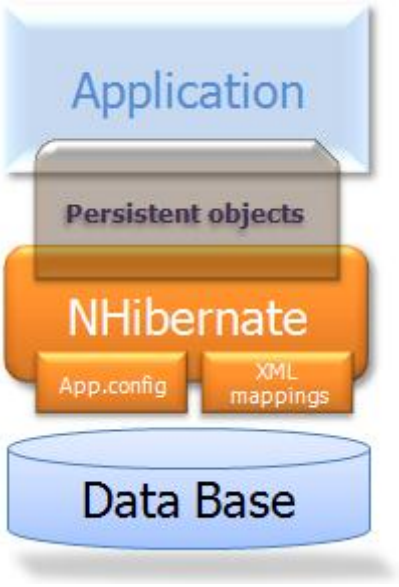


Figura 9. Arquitectura general

La figura mostra l'arquitectura general d'NH per a proporcionar serveis de persistència a l'aplicació. És difícil donar una vista més detallada en temps d'execució perquè NH és flexible i admet molts tipus d'enfocaments, per al que mostrarem els dos extrems. L'arquitectura bàsica té l'aplicació d'oferir les seves pròpies connexions ADO.NET i administrar les seves pròpies transaccions. Aquest model usa un subconjunt mínim de les API d'NH:

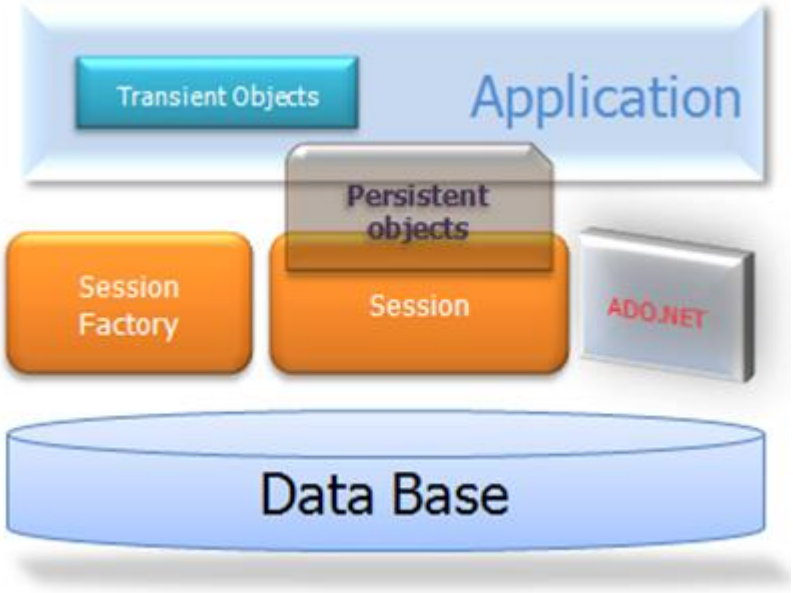


Figura 10. Arquitectura bàsica

L'arquitectura sencera permet tenir una visió més ample dels detalls:

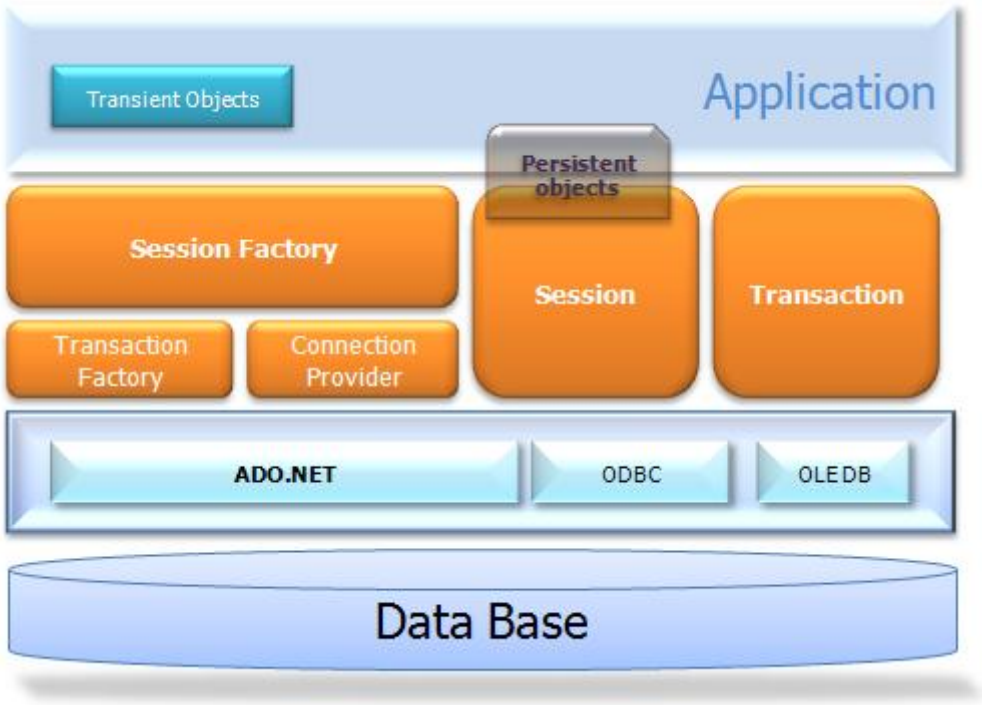


Figura 11. Arquitectura en detall

Passem a descriure cadascun dels objectes de l'arquitectura:

- **Session Factory** (NHibernate.ISessionFactory). Una memòria cau multi procés de mapatges compilats per a una única BBDD.
- **Session** (NHibernate.ISession). Objecte de curta durada d'un únic procés que representa una conversa entre l'aplicació i l'emmagatzematge persistent.
- **Objectes persistents i col·leccions**. Objectes de curta durada d'un únic procés que contenen l'estat persistent i la funcionalitat de negoci. Han d'estar associats amb una única sessió i no s'alliberaran fins que aquesta es tanqui.
- **Objectes transitoris i col·leccions**. Instàncies de classes persistents que no estiguin actualment associades amb cap sessió. Poden haver estat instanciades per l'aplicació i que encara no hagin estat persistides, o bé poden haver instanciades per una sessió tancada.
- **Transaccions** (NHibernate.ITransaction). Objecte de curta durada usat per l'aplicació per a especificar unitats atòmiques de treball. Abstrau l'aplicació subjacent d'una transacció ADO.NET.
- **Connection Provider** (NHibernate.Connection.IConnectionProvider). Proveïdor opcional de connexions i comandes d'ADO.NET. Abstrau l'aplicació d'haver de fer implementacions específiques del proveïdor.
- **Driver** (NHibernate.Driver.IDriver). Interfície opcional que encasella les diferències entre els diversos proveïdors ADO.NET.
- **Transaction Factory** (NHibernate..Transaction.ITransactionFactory). Transaccions opcionals no exposades per l'aplicació, però que poden ser implementades per al desenvolupador.

En arquitectures "lite" l'aplicació parla directament amb ADO.NET sense usar el Transaction Factory ni el Connection Provider.

2.5.6 Estats de les instàncies

Una instància d'una classe persistent pot tenir tres estats diferents, que es defineixen respecte a un estat de persistència. L'objecte ISession d'NH és el context de persistència:

- **Transitori**. La instància no està, ni mai ho ha estat, associada amb un context de persistència. No té identitat persistent (valor de la clau primària).
- **Persistent**. La instància està actualment associada amb un context de persistència. Té una identitat persistent i potser la seva corresponent fila a la base de dades.
- **Individual**. La instància va estar un cop associada amb un context de persistència, però o bé aquest context es va tancar, o bé la instància va ésser serialitzada a un altre procés. Té una identitat persistent i potser la seva corresponent fila a la BBDD.

2.5.7 Sessions contextuals

La majoria d'aplicacions que usen NH necessiten algun tipus de sessió contextual, on una sessió determinada està vigent en tot l'àmbit d'un context donat. No obstant, al llarg de les aplicacions la definició del que constitueix un context acostuma a ser diferent i contexts diferents defineixen diferents àmbits.

Des de la versió 1.2, NH va afegir el mètode ISessionFactory.GetCurrentSession(). Una interfície que l'estén (NHibernate.Context.ICurrentSessionContext) i un nou paràmetre de configuració (hibernate.current_session_context_class) s'han afegit per a permetre la connexió de l'àmbit i el context de la definició de sessions actual.

La versió 2.0.0 ja bé amb diverses implementacions d'aquesta interfície:

- NHibernate.Context.**ManagedWebSessionContext**. A les sessions en curs es realitza un seguiment per HttpContext.
- NHibernate.Context.**WebSessionContext**. De forma anàloga a l'anterior es guarda la sessió actual a HttpContext.
- NHibernate.Context..**CallSessionContext**. A les sessions en curs es realitza un seguiment per CallContext.
- NHibernate.Context..**ThreadStaticSessionContext**. La sessió en curs es guardada en una variable estàtica del procés.

El paràmetre de configuració "hibernate.current_session_context_class" defineix quina implementació de la interfície NHibernate.Context.ICurrentSessionContext s'usarà. Normalment, el valor d'aquest paràmetre seria el nom de la classe a usar (incloent el nom de l'assemblat).

2.5.8 Els models d'NH

Per a treballar amb NH, la forma més lògica de fer-ho seria seguint les següents passes:

1. Implementar a .NET les classes de domini persistents.
2. Crear a la base de dades les taules associades a cada classe.

3. Crear l'arxiu de mapatge de cada classe.
4. Crear l'arxiu de configuració d'NH.
5. Usar l'API d'NH per a persistir i recuperar els objectes

Un cop creats els models de domini a .NET i el model lògic a la BBDD relacional, el següent pas seria implementar els arxius de mapatge. NH, per a poder conèixer la correspondència existent entre els objectes i les taules, ho fa per mig de la configuració de mapeig. Aquesta configuració es pot fer o bé de forma programàtica, o bé la més usada, que consisteix en arxius de mapeig XML (mapping files). Aquests fitxers tenen la informació necessària per a poder saber a quina o quines taules s'han de guardar o rescatar cadascun dels objectes. Els noms dels arxius tenen el sufix .hbm.xml.

El següent seria un exemple d'arxiu de mapeig XML:

```
<?xml version="1.0" encoding="utf-8" ?>
        <hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="TFC.NH01"
        namespace="TFC.NH01.Entitats">

          <class name="Comanda">
                  <id name="Id" column ="Id Comanda " type="int" unsaved-value="0">
                    <generator class="identity"/>
                  </id>
                  <property name="Data" type="DateTime" not-null="true"/>
          </class>

      </hibernate-mapping>
```

Si ens fixem en l'estructura de l'exemple veurem que al començament de l'arxiu (a l'assemblat) indiquem on estaran ubicades les classes, seguit de l'espai de noms corresponent. Després segueixen les característiques de la classe a mapejar amb l'atribut "class" seguit del nom de la classe, i dins d'aquesta indicarem la configuració per a la classe en qüestió.

Amb la variable <id> s'indica a NH quina propietat es mapejarà amb la clau primària, en aquest cas "Id", després seguiria la resta de la configuració dels mapatges, relacions, etc. Val la pena apuntar també que un atribut important (que no està a l'exemple perquè bé per defecte) és lazy="true", que serveix per activar la càrrega vaga, de forma que les col·leccions no es carreguin fins que no hagin d'ésser usades.

Una pràctica molt recomanable i comú és incloure els arxius de mapeig com a recursos embeguts dins d'un assemblat. Qualsevol IDE actual, com el mateix VS, ens permet configurar les propietats dels arxius i quina acció cal fer amb ells. L'opció es diu "Action Build" i cal configurar-la com a "Embedded Resource". Si se'ns passa per alt es possible que el mapatge no funcioni correctament.

Anem ara a descriure unes petites observacions de com hem de preparar les classes. Primer, apuntar que treballant en el mode lazy="true", tots els mètodes i propietats han de ser declarats com "virtual", d'aquest mode NH pot crear un "proxy" de les nostres entitats. Segon, hi ha dos mètodes: Equals() i GestHashCode() que cal sobrescriure per a poder treballar correctament amb col·leccions. Finalment, dir que totes les entitats implementen

“IEquatable”, això no es necessari per a treballar amb NH però es una manera de fer que les comparacions amb Equals es facin de forma tipada.

2.5.9 Cadenes de connexió

Seguidament explicarem els conceptes d’”ISessionFactory” i d”ISession”, introduïts ràpidament quan hem descrit l’arquitectura. Una sessió és un marc de treball al qual NH estableix una conversa entre l’aplicació i el motor de la base de dades. Per a construir una sessió, representat per ISession el proveïdor de la sessió, necessitem també a ISessionFactory. Aquest darrer s’encarrega de crear sessions a la nostra aplicació, que pot tenir 1 o més sessions obertes. Cadascuna d’elles representa un primer nivell de memòria cau, on es dipositen els objectes que són portats des de la BBDD o són guardats a l’aplicació. Com a normal general no cal tenir un ISessionFactory per aplicació, només cal tenir-ne més d’un quan treballem amb més d’una base de dades simultàniament.

Per a configurar el ”ISessionFactory”, és a dir, per a dir-li: amb quin motor de BBDD treballarem, la cadena de connexió (connection string), el driver que usarem, etc., com tota la configuració a NH es pot fer de forma programàtica o bé amb arxius de configuració XML, i dins aquesta darrera opció ho podem fer mitjançant el ”App.config” o amb l’”hibernate.cfg.xml”. Un exemple d’aquest seria el següent:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-configuration  xmlns="urn:nhibernate-configuration-2.2" >
  <session-factory name="NH01">
        <property
name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>
        <property
name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
        <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
        <property name="connection.connection_string">Data
Source=localhost\SQLEXPRESS;Initial Catalog=NH01;Integrated Security=True</property>
        <property name="show_sql">true</property>
        <mapping assembly="TFC.NH01" />
  </session-factory>
</hibernate-configuration>
```

En aquesta configuració hem inclòs el nom de l’assemblat on es trobaran embeguts els arxius de mapeig. En aquest cas és: TFC.NH01.

El codi necessari per a configurar l’HN al començament de l’aplicació és:

```
Configuration cfg = new Configuration();
cfg.Configure("hibernate.cfg.xml");
ISessionFactory sessions = cfg.BuildSessionFactory();
ISession sessio = sessions.OpenSession();
```

Primer cal crear un objecte del tipus “Configuration”, configurar-lo amb la sentència “Configure” i fer que s’encarregui de crear l’”ISessionFactory” amb el mètode BuildSessionFactory(). Un cop que creem l’objecte sessions del tipus “ISessionFactory” ja podem començar a crear objectes “ISession”. En aquest cas es treballaria amb una sola

sessió a l'aplicació sessió. És una bona pràctica fer un "singleton" de l'objecte "ISessionFactory" degut a que és un procés costós per l'aplicació, per tant l'execució de BuildSessionFactory() cal fer-la una sola vegada durant l'execució.

2.5.10 Proveïdors de dades, consultes i transaccions

NH suporta les següents BBDD:

- Microsoft SQL Server
- Microsoft Access
- Oracle
- Firebird
- PostgreSQL
- DB2 UDB
- MySQL
- SQLite

2.5.11 Serialització amb NH

La serialització en NH té alguns problemes, que tractem seguidament.

Si optem per una serialització XML, ens podem trobar amb dificultats amb el serialitzador predeterminat de .NET, ja que no tracta les referències recursives i retorna per tant excepcions al trobar-les. Així doncs mentre aquest problema no estigui ben resolt és millor optar per la serialització binària.

Ja hem introduït anteriorment el concepte de les consultes vagues, que ara estendrem una mica més. Sense aquestes, NH al recollir un objecte de la BBDD i començar a formar-lo, es trobaria amb que tindria que recórrer diverses taules per a omplir els camps procedents d'altres claus alienes. Si aquests són alhora entitats d'altres taules, les quals també tenen altres claus alienes, podrem provocar una consulta enorme. Per tant, el resultat d'una consulta vaga seran entitats vagues o amb fills o col·leccions vagues: "lazy collections" que permeten a NH portar en una consulta només els atributs d'una taula, deixant les seves relacions en mode suspès fins que no calgui accedir-hi. Aquestes col·leccions són doncs molt recomanables d'usar per a optimitzar el rendiment de l'aplicació.

El problema que té serialitzar les "lazy collections" és que NH per a poder implementar-les als objectes del nostre domini, munta sobre les nostres interfícies "IList/IDictionay", col·leccions internes que contenen una referència oberta al moment de l'accés. A més, els objectes dels que només té informació sobre el seu identificador, per ser fruit d'una clau aliena, els insereix en una espècie de "proxy" per a que siguin refrescats al accedir a alguna de les seves propietats. Els inconvenients de tot això són:

- Una col·lecció "IList/IDictionay" que per dins implementa un tipus d'NH no podrà ser deserialitzada per una aplicació que no tingui accés a les llibreries d'NH.
- Una classe que té inserit un "proxy" no es pot serialitzar.

Aquests dos problemes tenen solució, però no es trivial. Per una banda, la solució al problema del "proxy" és usar una rutina d'NH anomenada "Unproxy", però té el problema de que només li treu el "proxy" a un objecte del nostre domini, però no als seus fills. Per altra banda, la solució al problema de les col·leccions és transformar-les a col·leccions natives de .NET (List i Dictionary) en temps d'execució, però ens trobarem amb que les col·leccions encara no recuperades no es podran convertir en col·leccions normals i haurem de marcar els objectes del nostre domini que estiguin incomplets.

2.6 LINQ to SQL

2.6.1 Introducció a LINQ

Abans de entrar en detall en LINQ to SQL, hem de fer una breu introducció a LINQ, acrònim de "Language Integrated Query", un projecte Microsoft per a la consulta i l'accés a dades, que bàsicament el que fa és agregar consultes natives similars a les de l'SQL als llenguatges de la plataforma .NET, inicialment amb VB.NET i C#.

LINQ defineix operadors de consulta estàndard que permeten als llenguatges: filtrar, enumerar i crear projeccions de diversos tipus de col·leccions usant la mateixa sintaxi. Aquestes poden incloure: vectors, classes enumerables, XML (LINQ to XML) i conjunts de dades provinents de bases de dades relacionals i d'orígens de dades de terceres parts.

L'objectiu de crear LINQ és permetre que tot el codi fet en VS (incloses les crides a BBDD, "datasets" o XML) siguin també OO. Abans de LINQ, la manipulació de dades externa tenia un concepte mes estructurat que no pas OO. A més, LINQ tracta de facilitar i estandarditzar l'accés a aquests objectes.

LINQ usa diverses característiques noves per a permetre als llenguatges l'ús de la sintaxi de consultes natives, com són: tipus anònims, mètodes extensors, expressions lambda ,arbres d'expressió o operadors de consulta estàndard.

Encara que LINQ suporta inicialment consultes en col·leccions a memòria, BBDD relacionals i dades XML, és una arquitectura extensible que permet a desenvolupadors d'orígens de dades addicionals l'ús de LINQ, implementant els operadors de consulta estàndard com mètodes extensors per als seus orígens de dades, o mitjançant la implementació de la interfície "IQuerytable", que permet convertir un arbre d'expressió en temps d'execució, per a transformar-lo en algun llenguatge de consultes. Els operadors de consulta estàndard son també usats per objectes i permeten consultar aquests a la memòria amb la mateixa sintaxi LINQ.

El framework LINQ inclou una eina anomenada SQLMetal, que permet la generació automàtica de classes directament des de una BBDD Microsoft SQL Server, permetent la integració de codi i base de dades de forma fàcil i ràpida. Aquesta generació de classes és totalment visual, només cal arrossegar les taules de la BBDD al codi generat per a realitzar la crida de les classes i posteriorment cridar a la funció "context" per a obtenir les dades de la classe.

2.6.2 Arquitectura de LINQ dins el .NET Framework

LINQ disposa d'una API amb un conjunt d'operadors estàndard de consultes. Els operadors suportats són: Select, Where, SelectMany, Sum / Min / Max / Average, Aggregate, Join / GroupJoin, Take / TakeWhile, Skip / SkipWhile, OfType, Concat, OrderBy / ThenBy, Reverse, GroupBy, Distinct, Union / Intersect / Except, SequenceEqual, First / FirstOrDefault / Last / LastOrDefault, Single, ElementAt, Any / All / Contains i Count.

Aquesta API també disposa d'operadors de conversió d'una col·lecció en un altre tipus de dades, com són: AsEnumerable, To Queryable, ToArray, ToList, ToDictionary, ToLookup, Cast i OfType.

Encara que LINQ està implementat com una llibreria per al .NET Framework 3.5, també defineix una sèrie d'extensions del llenguatge, les quals inclouen: consultes sintàctiques, variables tipades implícites, tipus anònims, inicialitzadors d'objectes i expressions lambda.

LINQ ve amb els proveïdors de LINQ per a col·leccions d'objectes en memòria, BBDD d'SQL Server, "datasets" d'ADO.NET i documents XML. Aquests diversos proveïdors defineixen les diferents implementacions de LINQ:

- **LINQ to Entities**, proveïdor usat per a fer consultes sobre col·leccions en memòria, usant el motor de LINQ per l'execució de consultes. El codi generat per aquest proveïdor es refereix a la implementació dels operadors de consulta estàndard tal i com es defineix al patró de seqüència i permet col·leccions "IEnumerable" per a ser consultades a nivell local.
- **LINQ to XML** (XLINQ), proveïdor que converteix un document XML a una col·lecció d'objectes de tipus "XElement", els quals són consultats contra el motor d'execució local proporcionat com a part de la implementació de l'operador de consulta estàndard.
- **LINQ to SQL** (DLINQ), el proveïdor que estudiarem tot seguit més en detall, permet a LINQ fer consultes contra bases de dades d'SQL Server.
- **LINQ to DataSets**, com el proveïdor LINQ to SQL treballa només amb Microsoft SQL Server, per a donar suport a qualsevol altre BBDD genèrica, LINQ també inclou la LINQ to Dataset, que usa ADO.NET per comunicar amb la base de dades. Un cop les dades estan en conjunts de dades ADO.NET, LINQ to DataSet executa consultes contra aquests DataSets.
- **Altres proveïdors**, que poden ser implementats per terceres parts per a diverses bases de dades. N'hi ha forces disponibles, com ara: Data Services, dotConnect, EF, NH, DbLinq, CSV, etc.

2.6.3 Introducció a LINQ to SQL

LINQ to SQL és un framework ORM que permet el mapatge directe 1:1 d'una base de dades Microsoft SQL Server a les classes de .NET, i la consulta dels objectes resultants usant LINQ. Més específicament, LINQ to SQL ha estat desenvolupat per a arribar a un escenari de ràpid desenvolupament contra SQL Server, on la BBDD s'assembla molt al model d'objectes de l'aplicació, deixant així que la principal preocupació del desenvolupador pugui ser centrar-se únicament en incrementar la productivitat.

LINQ to SQL ha estat dissenyat amb la simplicitat i la productivitat del desenvolupador en ment. Les seves API estan dissenyades per començar a treballar ràpidament en escenaris d'aplicacions comunes. Exemples d'aquest disseny inclouen: la capacitat de reemplaçar les convencions de noms no amigables de la BBDD amb noms descriptius, mapejar l'esquema dels objectes directament a les classes de l'aplicació, carregar les dades de forma implícita encara que no hagin estat prèviament carregades a la memòria, etc.

Com hem esmentat anteriorment, l'objectiu de LINQ to SQL és proporcionar escenaris on la base de dades sigui més propera al model d'objectes de l'aplicació. Davant això, la forma de traduir l'esquema SQL i les classes de l'aplicació és un mapatge 1:1, que significa que una taula o vista de la BBDD és mapejada a una classe senzilla de l'aplicació, i una columna a una propietat de la classe associada.

Podríem finalment definir LINQ to SQL com al component específic de LINQ que proporciona la infrastructura en temps d'execució necessària per a utilitzar dades relacionals com a objectes i poder definir consultes sobre aquests. És a dir, habilita la consulta de contenidors de dades relacionals sense tenir que abandonar la sintaxi o l'entorn de temps de compilació.

2.6.4 Principals característiques

Principals avantatges de LINQ to SQL:

- L'entorn visual de creació del model (ORD).
- Mapatge 1:1 entre la BBDD i les classes, propietats, mètodes, etc., de l'aplicació.
- Permet jerarquies d'herència que poden ser emmagatzemades en una senzilla taula.
- Integració de la informació de l'esquema de la base de dades en metadades del CLR, que ens permet usar les nostres pròpies classes CLR sense format enlloc d'utilitzar les classes derivades o generades d'una classe base o interfície.
- Optimitza el rendiment gràcies als procediments emmagatzemats i a les consultes compilades.
- La persistència, que habilita el control automàtic de canvis a la BBDD, i l'actualització de dades per mig de sentències T-SQL.

2.6.5 Arquitectura i accés a dades

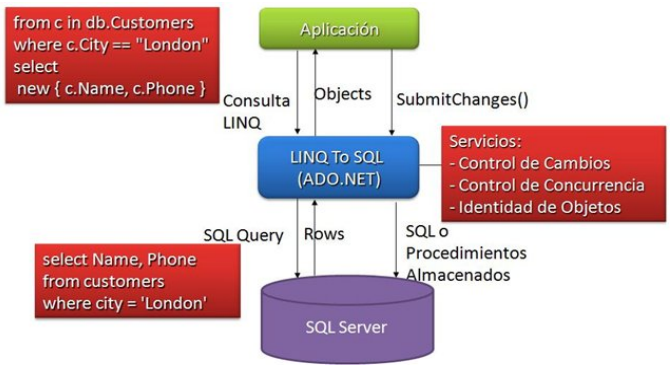


Figura 12. Arquitectura

L'esquema de la imatge superior és l'arquitectura de LINQ to SQL. Com es pot deduir els punts claus que té són els següents:

- Consultes a nivell d'aplicació integrades al llenguatge.
- BBDD tipada en objectes: mapeig per mig d'atributs, retorna objectes al realitzar consultes SQL.
- Es garantitza la persistència, control de canvis i de concurrència.
- Es tradueixen les consultes SQL que s'envien a la base de dades: les consultes no s'executen a memòria i no es transmeten com "IL" al servidor.

2.6.6 Els models

Com les dades d'SQL Server poden residir en un servidor remot i aquesta BBDD té el seu propi motor de consulta, LINQ to SQL no usa el motor de LINQ. En el seu lloc, converteix una consulta LINQ per a una consulta SQL que s'envia a l'SQL Server per al seu processament. Com l'SQL Server és una BBDD relacional i LINQ treballa amb dades encapsulades en objectes, les dues representacions s'han de mapejar, per aquesta raó LINQ to SQL defineix un framework ORM.

El mapatge es fa definit classes que corresponen a taules de la BBDD i contenen la totalitat o un subconjunt de les columnes de la taula com a membres de dades. La correspondència, juntament amb altres atributs del model relacional com la clau primària, són especificats utilitzant els atributs definits a LINQ to SQL.

El següent és un clar gràfic representatiu de com es fa el mapatge:

Figura 13. Mapatge

Les classes han de ser definides abans d'usar LINQ to SQL. VS 2008 inclou un dissenyador de mapatges que pot ser utilitzat per a crear-los. Permet crear automàticament les classes des d'un esquema de BBDD, així com permetre l'edició manual per a crear diferents vistes usant només un subconjunt de les taules o columnes d'una taula.

El mapatge és implementat mitjançant el "DataContext", el qual proporciona una cadena de connexió al servidor i pot ser usat per a generar una "Table<T>", on T és el tipus al qual es mapejarà la BBDD. Aquesta "Table <T>" encasella les dades a la taula i implementa la interfície "IQueryable<T>", on es crea l'arbre d'expressió amb el qual treballa el proveïdor de LINQ to SQL. Aquest converteix la consulta a T-SQL i recupera el conjunt de resultats del servidor de BBDD. Atès que el processament ocorre en aquest, els mètodes locals, que no es defineixen com a part de les expressions lambda que representen els predicats, no es poden utilitzar. No obstant, pot utilitzar els procediments emmagatzemats al servidor. Qualsevol canvi en el conjunt de resultats es seguit i es pot enviar de tornada al servidor.

Seguidament detallarem dels principals passos a seguir per a implementar una aplicació de LINQ to SQL. Alguns d'ells són opcionals, és possible usar el model d'objectes en el seu estat per defecte.

El primer pas és crear el model d'objectes. Per a aquesta funcionalitat hi ha tres eines disponibles:

- **El dissenyador d'objectes relacional** (ORD), que proporciona una rica interfície visual per a crear objectes LINQ to SQL i associacions basades en objectes de la base de dades. Aquesta eina és part de l'IDE de VS i s'adapta millor a les BBDD petites i mitjanes.
- **L'eina de generació de codi SQLMetal**, utilitat de línia de comandes que proporciona un conjunt d'opcions lleugerament diferent de l'anterior. El modelatge de bases de dades grans es fa millor usant aquesta eina.
- **Un editor de codi**, podem escriure el nostre propi codi usant l'editor de VS o qualsevol altre, però no es una opció recomanable perquè és molt fàcil cometre errades. En tot cas es millor usar-lo per acabar de redefinir el model ja creat per alguna de les dues opcions anteriors.

El segon pas seria seleccionar el tipus de codi a generar, que pot ser:

- Un fitxer de codi en C# o Visual Basic.NET, per a un mapatge basat en atributs.
- Un fitxer XML per a un mapatge extern. Amb aquest enfocament es pot evitar l'entrada de metadades de mapeig de fora del codi de l'aplicació. Val a dir que l'ORD no suporta aquest tipus, si el volem implementar haurem d'usar l'SQLMetal.
- Un fitxer DBML, que es pot modificar abans de generar el codi final.

El tercer i darrer seria redefinir el fitxer de codi generat, per a reflectir les necessitats de la nostra aplicació. Per a aquest objectiu es pot usar o bé l'ORD o bé l'editor de codi.

La següent imatge mostra la relació entre el desenvolupador i les dades:

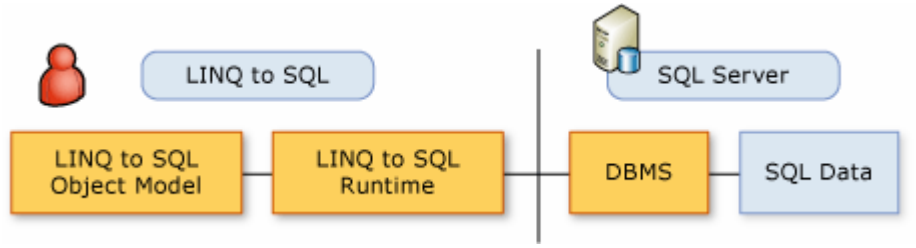


Figura 14. Relació entre desenvolupador i dades

Un cop tenim el model d'objectes cal descriure les sol·licituds d'informació i manipular les dades dins d'aquest model. Hem de pensar en termes d'objectes i propietats del model d'objectes, i no en termes de les files i columnes de la base de dades, ja que no tractarem directament amb aquesta.

En instruir LINQ to SQL per a executar una consulta descrita o per a cridar al mètode SubmitChanges() a les dades manipulades, LINQ to SQL es comunica amb la base de dades en el llenguatge d'aquesta.

Els següents són els passos típics per a usar el model que hem creat:

1. Crear les consultes per a extreure la informació de la BBDD.
2. Sobrescriure els mètodes per defecte d'inserir, actualitzar i esborrar.
3. Configurar les opcions adequades per a detectar i informar de conflictes de concurrència. Podem deixar els valors per defecte del nostre model o bé els podem canviar.
4. Establir una jerarquia d'herència.
5. Proporcionar una interfície d'usuari apropiada. Aquest pas és opcional.
6. Testejar l'aplicació.

2.6.7 Cadenes de connexió

Un cop creat el model d'objectes, cal definir el canal que ens permeti portar els objectes des de la base de dades i bolcar-hi els canvis realitzats. Aquest canal és el que en LINQ es coneix com a "DataContext", que s'encarrega de traduir les peticions d'objectes a consultes SQL i retornar els resultats com a objectes.

El DataContext s'usa igual com si es tractes d'un objecte "SqlConnection" d'ADO.NET. El seu propòsit es convertir les sol·licituds d'objectes en consultes SQL que s'executin a la BBDD, i a continuació, assemblar els objectes a partir dels resultats. DataContext habilita LINQ al implementar el mateix model d'operador que els operadors de consulta estàndard, com Where i Select. El DataContext es pot especificar de dues maneres:

La primera es definint una instància del mateix i passant com a paràmetre la cadena de connexió de la base de dades. Aquest mètode implica que per a definir consultes haguem d'utilitzar el mètode GetTable(). Posem un exemple:

```
DataContext BD=new DataContext(@"C:\Program Files\LINQ Preview\Data\CLIENTS.mdf");
```

```
Table<Clients> Customers=BD.GetTable<Clients>();
```

En aquest exemple hem vist que com a cadena de connexió hem especificat la ruta física on tenim emmagatzemada la base de dades. El DataContext però, admet tres sobrecàrregues (una és la típica cadena de connexió d'ADO.NET) a la definició de la cadena de connexió.

Posarem un altre exemple que usa el DataContext per a establir una connexió amb la BBDD d'exemple de Northwind i recuperar les files de clients que estiguin a Londres:

```
'DataContext takes a connection string.
Dim db As New DataContext("…\Northwnd.mdf")
'Get a typed table to run queries.
Dim Customers As Table(Of Customer) = db.GetTable(Of Customer)()
'Query for customer from London.
Dim Query = _
   From cust In Customers _
   Where cust.City = "London" _
   Select cust
For Each cust In Query
   Console.WriteLine("id=" & cust.CustomerID & _", City=" & cust.City)
Next
```

La segona és definint l'objecte DataContext fortament tipat, fet que facilita la definició de les consultes a la base de dades, ja que en aquest cas no cal usar GetTable després de crear una instància d'aquest objecte. En aquest cas, cal crear una classe que hereti de DataContext.

Un cop especificat el DataContext ja es poden començar a realitzar consultes contra el model d'objectes.

2.6.8 Consultes i transaccions

Com hem comentat, un cop definit el model i especificat el DataContext, ja podem consultar i obtenir fàcilment dades de la nostra base de dades. Per això hem d'usar la sintaxi de consultes de LINQ, que té com a gran virtut que és idèntica sobre qualsevol tipus de dades.

La sintaxi de consultes és un conveni de declaracions per a expressar consultes usant els operadors estàndard de LINQ. Aporta una sintaxi que augmenta la claredat alhora d'escriure consultes al codi i pot ésser més fàcil de llegir i escriure correctament. VS incorpora un "intellisense" i reconeixement sistemàtic en temps de compilació per a la sintaxi de consultes.

El següent és un exemple de consulta:

```
NorthwindDataContext db = new NortWindDataContext();
Var products = from in db.Products
                    Where p.CategoryID == 2
                    Select p;
Foreach (Product product in products)
{
        product.ProductID}
```

En ella hem usat la sentència “where” per a retornar els productes d’una categoria. Estem usant el camp/propietat “CategoryID” del producte per a fer el filtre.

Un dels avantatges que proporciona LINQ to SQL és que ens dona una gran flexibilitat de com fer les consultes a les nostres dades, i podem aprofitar-nos de les associacions que haurem fet durant l’etapa de modelatge de les classes per a fer consultes més naturals i riques sobre la BBDD.

Sobre les transaccions, ja hem comentat anteriorment que són serveis per a garantir que una sèrie d’accions individuals puguin ocórrer de forma automàtica, el que significa que totes elles seran o bé un èxit o bé un fracàs. Si no hi ha cap transacció ja en l’àmbit de l’aplicació, quan cridem al mètode SubmitChanges() el DataContext iniciarà automàticament una transacció a la base de dades que guardi les actualitzacions.

Podem triar a controlar: el tipus de transacció a usar, el seu nivell d’aïllament, o el seu abast, inicialitzant-la nosaltres mateixos. L’aïllament de la transacció que usa el DataContext és conegut com a “ReadCommitted”. LINQ to SQL suporta tres models de transaccions diferents, que enumerem a continuació:

1. **Transacció local explícita**. Quan es crida SubmitChanges() si la propietat Transaction es posa a (IDbTransaction) transacció, el mètode SubmitChanges() s’executa en el context de la mateixa transacció. És la nostra responsabilitat de confirmar o desfer la transacció després de l’execució correcta d’aquesta. La connexió corresponent a la transacció ha de coincidir amb la connexió usada per a construir el DataContext, en cas contrari es produiria una excepció.
2. **Transacció distribuïda explícita**. Podem cridar a les API de LINQ to SQL (incloent però no limitat a SubmitChanges()) en l’àmbit d’una transacció activa. LINQ to SQL detecta que la crida està dins l’àmbit d’una transacció i no en crea una de nova. En aquest cas també evita el tancament de la connexió. Es poden executar consultes i crides a SubmitChanges() en el context d’aquesta transacció.
3. **Transacció implícita**. Quan es crida SubmitChanges() LINQ to SQL comprova si la crida és dins l’àmbit d’una transacció o si la propietat Transaction (IDbTransaction) està configurada com “user-started local transaction”. Si no troba cap transacció, LINQ to SQL inicia una transacció local i l’usa per a executar les comandes SQL generades. Quan totes aquestes comandes han finalitzat correctament, LINQ to SQL confirma la transacció local i retorna.

2.6.9 Serialització

Es pot serialitzar el codi en temps de disseny per a qualsevol dels dos mètodes següents:

- Amb l’**ORD**, canviant la propietat “Serialization Mode” a unidireccional.
- Amb la utilitat de línia de comandes de l’**SQLMetal**, afegint l’opció “/serialization”.

El codi generat per LINQ to SQL proporciona capacitats de càrrega ajornada per defecte. Aquesta opció és molt útil en el nivell mitjà de càrrega transparent de dades sota demanda, però és problemàtica per a la serialització, pel fet que els disparadors d’aquesta ajornen la càrrega si la càrrega diferida és intencionada o no.

Les característiques de la serialització de LINQ to SQL solucionen aquest problema per mitjà de dos mecanismes:

- El mode DataContext per desactivar la càrrega ajornada (ObjectTrackingEnabled).
- Un interruptor de generació de codi per a generar atributs del tipus System.Runtime. Serialization. DataContractAttribute i .DataMemberAttribute als objectes generats. Aquest aspecte, incloent el comportament de les classes de càrrega ajornada sota la serialització, és l'aspecte principal d'aquest apartat.

Definim ara un parell de conceptes, encara que la serialització unidireccional és l'únic tipus suportat per LINQ to SQL:

- **Serializador DataContract**. Serializador per defecte usat pel Windows Communication Framework (WCF), component del .NET Framework 3.0 o versions posteriors.
- **Serialització unidireccional**. Versió serialitzada d'una classe que conté només una propietat d'associació en un sol sentit (per evitar un cicle). Per conveni, la propietat a la banda dels pares d'una relació de clau primària és marcada per la serialització. L'altra banda en una associació bidireccional no és serialitzada.

Per acabar aquest apartat descriurem molt breument com convertir els objectes a serialitzables. Això es fa quan generem el codi. Les classes de les entitats han de ser configurades amb l'atribut DataContractAttribute, i les columnes amb l'atribut DataMemberAttribute. Per a aquest objecte, novament es poden usar les utilitats ORD o SQLMetal.

3 Síntesi de l'anàlisi, disseny i implementació

3.1 Introducció i requeriments

El punt de partida de qualsevol especificació és formular el problema que cal resoldre amb el programari en termes del client que ens l'encarrega. Hem de tenir ben present la pregunta: què ha de fer i oferir aquest programari? Cal posar molta atenció en la identificació dels requeriments i dels subsistemes que caldrà desenvolupar.

En el nostre cas el problema que volem resoldre és poder persistir objectes, construïts en el món conceptual de la nostra aplicació sobre una base de dades relacional. També voldrem fer l'operació contraria, és a dir: rescatar dades de la BBDD i tornar-les a l'aplicació en forma d'objectes.

Així, per una banda tenim com a únics requeriments funcionals, que cal desenvolupar una aplicació que ens permeti resoldre el problema que acabem de descriure. Per altra banda, com a requeriments no funcionals, el TFC ens demana que treballem amb unes eines determinades de Microsoft com són la plataforma Visual Studio i el SGBD SQL Server. Finalment el projecte seleccionat ens demanava escollir uns frameworks que implementessin el patró ORM, que en aquest cas seran: Entity Framework, NHibernate i LINQ to SQL.

Veiem doncs, que en aquest cas el document d'especificació a generar pot ser molt breu, doncs a diferència de la gran majoria d'aplicacions no és el client qui ens demana unes funcionalitats en concret, sinó que sens dona llibertat de criteri. Som per tant nosaltres mateixos qui hem proposat una senzilla aplicació que compleixi aquests únics requeriments a complir. Tot i així hem generat igualment un petit document especificatiu, on hem inclòs els següents apartats:

- Descripció del sistema.
- Descripció del procés.
- Resum esquemàtic de totes les funcionalitats del programari.
- Identificació dels subsistemes.
- Recursos necessaris.

3.2 Especificacions

L'aplicació que hem desenvolupat és una petita base de dades d'una escola a la qual hi ha professors i alumnes. Els primers generaren cada cert temps exàmens, que han de ser realitzats pels alumnes, i supervisats pels mateixos professors.

El sistema té 4 classes bàsiques:

- **Persona**.
- **Professor**, subclasse que implementa una herència per especialització de la superclasse Persona.

- **Alumne**, subclasse del mateix tipus que l'anterior.
- **Examen**, que té un atribut "nota".

També té 3 relacions:

- **Realitza (binària)**, entre les classes Alumne i Examen. Un alumne pot realitzar 1:N exàmens i un examen pot ser realitzat per 1:N alumnes.
- **Genera (binària)**, entre les classes Professor i Examen. Un professor pot generar 1:N exàmens i un examen pot haver estat generat per un sol professor.
- **Supervisió (ternària)**, entre les classes Examen, Alumne i Professor. Per cada examen, 1 professor pot supervisar a 1:N alumnes i 1 alumne pot tenir 1 sol professor.

El programari disposa d'una pantalla amb 4 camps a la part superior:

- Examen.
- Alumne.
- Professor.
- Nota.

A la part central hi ha una llista de tots els exàmens realitzats a l'escola i a sota té unes comandes RadioButton per a permetre seleccionar entre els següents tipus d'accés a dades:

- **ADO.NET**, que tot i no entrar en l'estudi hem afegit per a veure com es fa l'accés a dades tradicionalment en .NET quan no s'implementa cap framework ORM.
- **Entity Framework**, primer framework ORM de l'estudi.
- **NHibernate**, segon framework ORM de l'estudi.
- **LINQ to SQL**, tercer framework ORM de l'estudi.

Finalment, a la part inferior hi ha 5 botons amb les següents funcionalitats:

- **Connectar**. Realitzar la connexió amb la BBDD.
- **Netejar**. Esborrar les dades en pantalla, dels camps superior i del llistat.
- **Afegir**. Aquest botó té una doble funcionalitat. Per una banda, en el cas de no haver rescatat prèviament les dades de la llista, si s'omplen manualment tots els camps en pantalla, el sistema permet afegir a la BBDD les noves dades introduïdes, sempre que la clau primària sigui diferent a cap de les ja existents a la base de dades. Per altra banda, en el cas de si haver rescatat dades del "Listbox" però haver modificat alguns dels dos camps de la clau primària (examen i alumne) el sistema també permet afegir les noves dades a la BBDD.
- **Actualitzar**. En cas d'haver rescatat prèviament les dades de la llista sobre els camps superiors, aquestes es poden modificar per actualitzar-les posteriorment a la base de dades, sempre que cap dels dos camps de la clau primària no s'hagin modificat.
- **Eliminar**. Seleccionant un examen del llistat, l'aplicació permet eliminar-la de la base de dades.

Els subsistemes es poden identificar fàcilment donada la simplicitat de l'aplicació i el seu objectiu, que no és altre que realitzar l'accés a dades amb les tres tecnologies estudiades més

ADO.NET. Al construir l'aplicació seguint el model de programació per capes, aquests són els següents:

- **Subsistema de presentació**: interfície gràfica (GUI).
- **Subsistema de negoci**: funcionalitats.
- **Subsistema comú**: entitats de l'aplicació (persona, alumne, professor, examen i supervisa).
- **Subsistema de dades**: Accés a dades amb les diferents tecnologies: ADO.NET, Entity Framework, NHibernate i LINQ to SQL.

Per a fer aquesta aplicació hem emprat els següents recursos, a banda dels ja descrits a l'apartat 1.5:

- **Eina CASE**: ArgoUML.
- **Programari específic ORM**: Entity Framework, NHibernate i LINQ to SQL.

Finalment, és important comentar que enlloc de fer N implementacions diferents (una per tecnologia estudiada) hem preferit donar una solució integrada i fer-ne una de sola, amb una selecció a la capa de presentació que permet escollir una de les quatre tecnologies d'accés a dades. A la capa de negoci és on es crida a la classe que correspongui de la capa de dades. Val a dir que això ha complicat força l'aplicació i que si hores d'ara l'haguéssim de tornar a fer, probablement la plantejaríem separada.

3.3 Model estàtic

3.3.1 Model ER

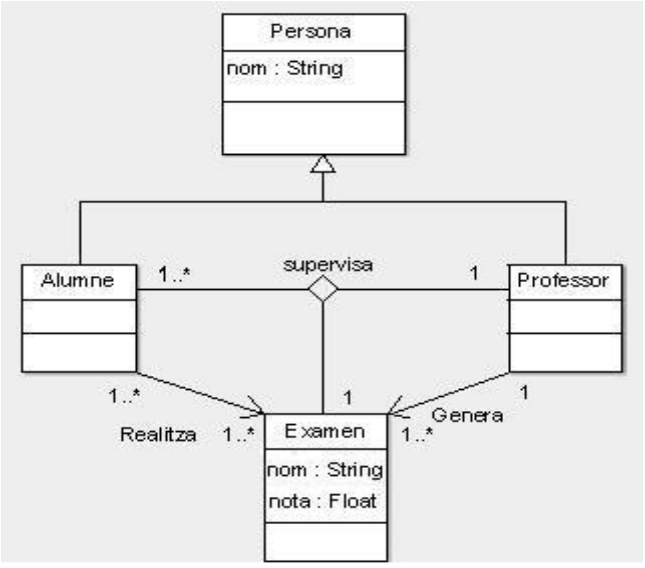


Figura 15. Model conceptual

El model estàtic d'UML consta, per una banda, de classes i objectes, i de l'altra, de relacions de diferents menes entre aquests. En la nostra aplicació, el model conceptual és el mostrat a la figura superior.

3.3.2 Diagrama de classes

El següent pas alhora de dissenyar el model estàtic és la transformació entre el model ER i el diagrama de classes.

Els llenguatges de programació no entenen conceptes com ara classe associativa o associacions n-àries, motiu per al qual si en el nostre model conceptual tenim algun d'aquests elements caldrà transformar-lo a una classe senzilla, procés que s'anomena reificació i que substitueix la classe associativa per una nova classe amb els mateixos atributs.

En la nostra aplicació no tenim cap classe associativa però si que tenim una relació ternària que cal transformar de la mateixa manera. La clau primària de la nova relació està formada pels atributs amb la clau primària de les entitats interrelacionades.

Finalment, les darreres transformacions que cal fer. La superclasse Persona l'eliminem, deixant només les seves subclasses. També cal substituir totes les associacions per atributs a les classes relacionades. Així, el diagrama queda com es mostra a la imatge inferior.

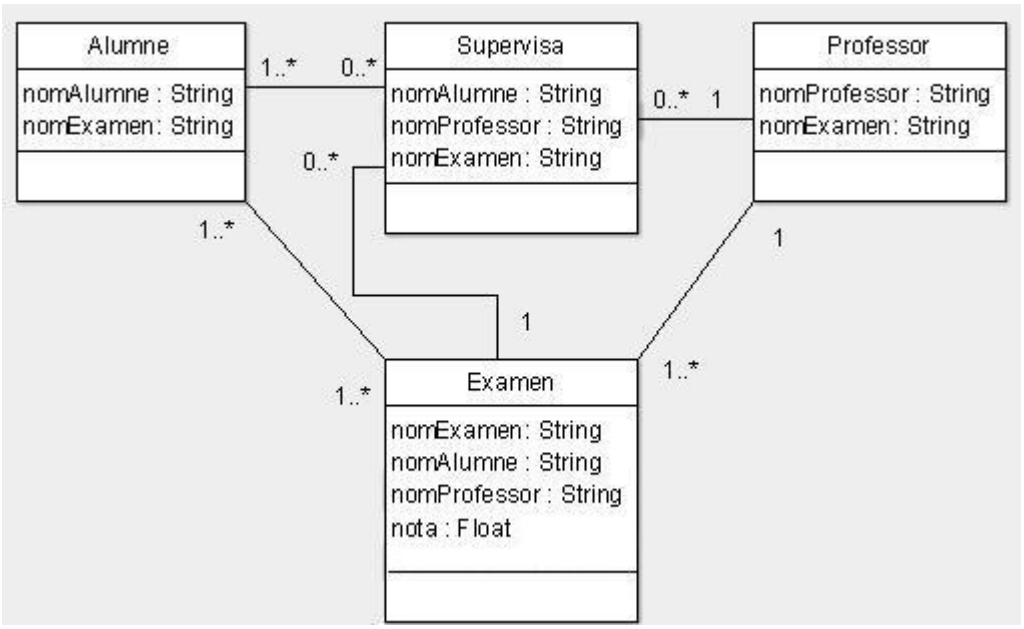


Figura 16. Diagrama de classes

3.4 Model dinàmic

3.4.1 Introducció

El model dinàmic i d'implementació, típicament es composa dels diagrames de:

- **Estats**, quan hi ha objectes que poden variar al llarg del temps, en aplicacions seqüencials i/o en que els objectes tenen estats.

- **Casos d'ús**, que serveixen per a mostrar les funcions d'un programari des del punt de vista de les seves interaccions amb l'exterior, però sense entrar en la descripció detallada ni en la seva implementació.
- **Iteració**, l'especificació del comportament dels casos d'ús o operació en termes de seqüències d'intercanvi de missatges entre objectes.
- **Activitats**, les quals serveixen per a descriure els estats d'una activitat.
- **Implementació**, que serveixen no per descriure la funcionalitat del programari sinó la seva estructura general amb vista a la seva construcció, execució i instal·lació. Té dues parts: el diagrama de components i el de desplegament.

En el nostre cas però, que es redueix a una senzilla aplicació d'accés a dades, no són necessaris la majoria d'aquests diagrames i ens limitem a descriure els casos d'ús principals.

3.4.2 Diagrames de casos d'ús

Un cas d'ús documenta una interacció entre el programari i un actor o més. Ha de ser en principi una funció autònoma dins el programari. En la nostra aplicació hem previst un sol actor: l'usuari del sistema, que serem nosaltres mateixos ja que aquest no és un programari per ésser implementat enlloc, sinó un treball pràctic sobre les diverses tecnologies estudiades.

Els casos d'ús de la nostra aplicació són els següents:

1. Connectar.
2. Afegir.
3. Actualitzar.
4. Eliminar.

3.4.3 Cas d'ús Connectar

Aquesta funció pot diferir força en funció de quina de les quatre tecnologies estiguem executant, però a grans trets el funcionament és similar. Per exemple, amb ADO.NET, quan premem el botó Connectar, primer s'executa l'objecte "Connection" per a definir de quina manera i a quina base de dades ens connectarem, i seguidament l'objecte "Command" per a preparar l'accés a les dades, executant la comanda SQL "Select * from Examens".

Posteriorment implementem l'objecte "DataReader" per a transportar les dades des de la taula Examens de la nostra BBDD relacional a la classe Examen de l'aplicació, que hem creat prèviament amb els mateixos atributs que columnes té la taula. Tot seguit amb el mètode "ToSTring()" d'aquesta classe, la llegim i escrivim sobre la llista de la nostra interfície, per a mostrar tots els atributs de la classe Examen.

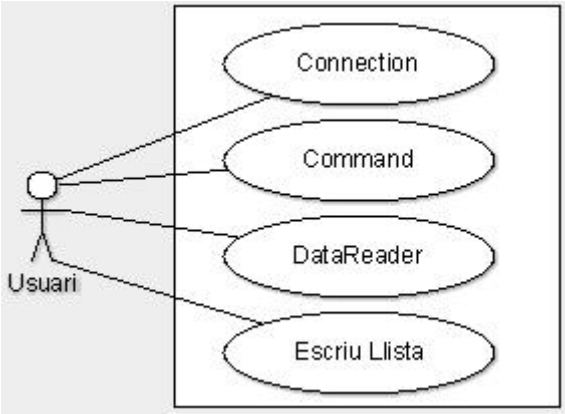


Figura 17. Cas d'ús Connectar

3.4.4 Cas d'ús Afegir

Aquesta funcionalitat de fet són dues en una, ja que ens permet o bé generar noves dades a la BBDD, o bé actualitzar les rescatades prèviament en cas d'haver modificat algun dels camps de la clau primària.

En el primer cas, sense rescatar les dades prèviament de la llista, hem d'omplir els camps en pantalla amb les noves dades. En el segon, primer cal rescatar les dades i després modificar com a mínim un dels dos camps que componen la clau primària (examen o alumne). Els altres dos camps es poden modificar o deixar com estaven.

En qualsevol dels dos casos, al prémer el botó d'afegir s'afegeixen les noves dades a la BBDD executant la comanda SQL "`INSERT INTO Examen (nomExamen, nomAlumne, nomProfessor, nota) values` (examen, alumne, professor, nota)", sempre que la clau primària no sigui ja existent. Posteriorment es netegen les dades dels camps en pantalla.

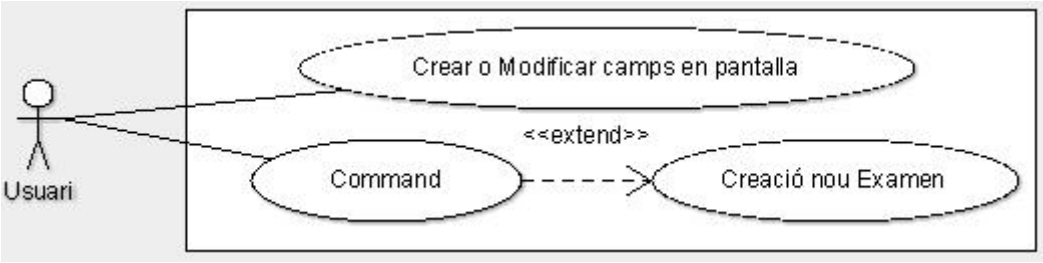


Figura 18. Cas d'ús Afegir

3.4.5 Cas d'ús Actualitzar

Aquesta funció ens permet actualitzar les rescatades prèviament de la llista. Un cop fetes les modificacions als camps de la pantalla, al prémer sobre el botó actualitzar, s'actualitzen les dades a la BBDD executant la comanda SQL "`UPDATE Examen SET nomExamen` = Examen, ..." sempre que no s'hagi modificat cap dels dos camps de la clau primària. Posteriorment es netegen també les dades dels camps en pantalla.

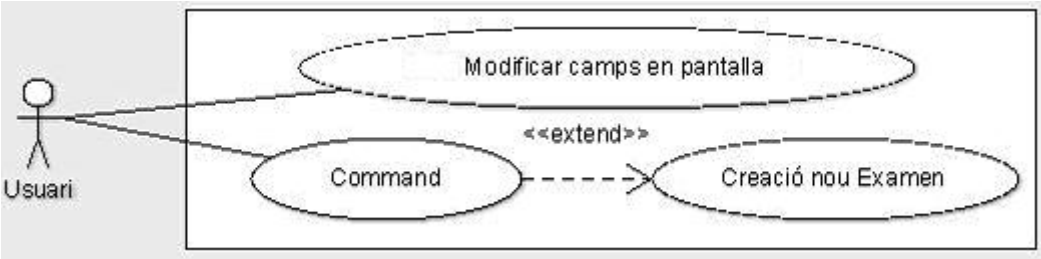


Figura 19. Cas d'ús Actualitzar

3.4.6 Cas d'ús Eliminar

Aquesta funcionalitat permet esborrar la fila seleccionada de la llista a la seva fila corresponent de la BBDD. Un cop feta la selecció, executem la comanda SQL "Delete * from Examens where Examen = x and Alumne = y", on "x" és el nom de l'examen i "y" el nom de l'alumne de la línia seleccionada a la llista. Posteriorment es netegen novament les dades dels camps en pantalla.

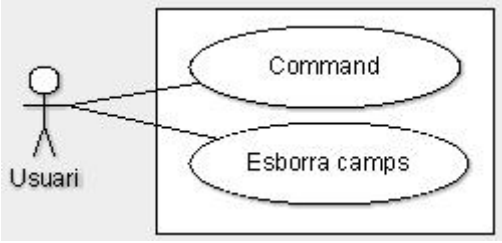


Figura 20. Cas d'ús Eliminar

3.5 Disseny de l'aplicació

3.5.1 Disseny de la base de dades

Partint del diagrama de classes creat a l'apartat 2.2, obtenim el disseny de les taules de la nostra BBDD:

- **Alumne**(nomAlumne, nomExamen);
- **Professor**(nomProfessor, nomExamen);
- **Examen**(nomExamen, nomAlumne, nomProfessor, nota);
- **Supervisa**(nomAlumne, nomProfessor, nomExamen);

3.5.2 Diagrama de la base de dades

Mostrem seguidament el diagrama de classes i les seves relacions implementades directament sobre l'SQL Server 2008:

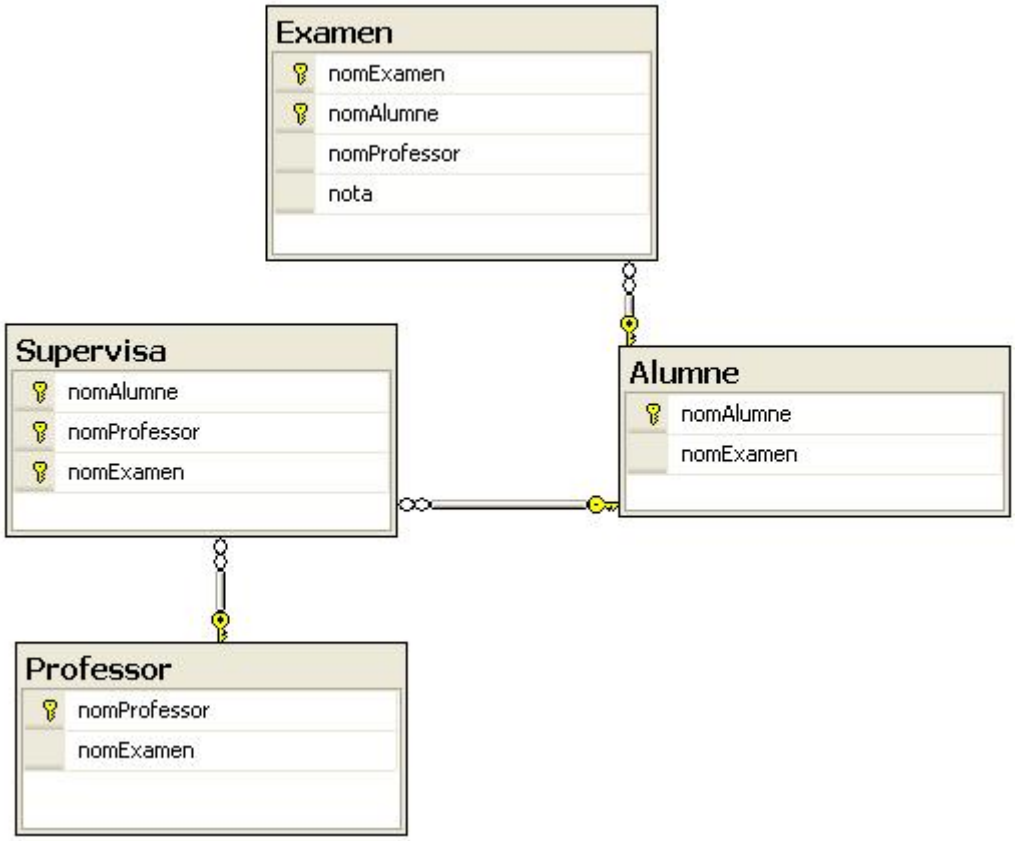


Figura 21. Diagrama de classes a l'SQL Server 2008

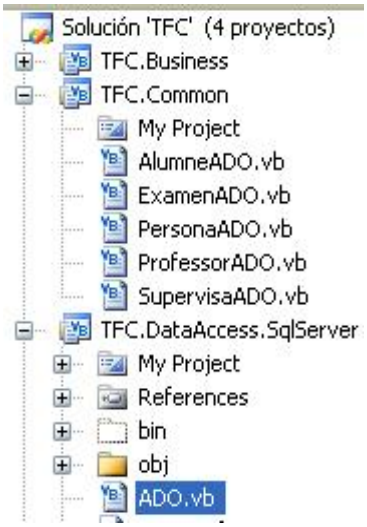
3.5.3 Interfície gràfica

Alhora de dissenyar la interfície gràfica s'ha tingut en compte a qui va dirigida, que en aquest cas seria un usuari experimentat, i especialment s'ha intentat garantir en tot moment la usabilitat de l'aplicació, fent que aquesta sigui el màxim de senzilla i eficient possible, i aplicar els principis bàsics de disseny gràfic per a fer la interfície usable:

- Agrupament.
- Visibilitat.
- Consistència.
- Economia del disseny.
- Color com a suplement.
- Reducció del desordre.

3.6 Implementació

3.6.1 Implementació d'ADO.NET



La implementació d'ADO.NET està fora de l'abast del pla de treball del nostre projecte final de carrera, però hem considerat oportú afegir-la, per a poder comparar d'una forma pràctica la diferencia entre treballar amb patrons ORM, o bé directament de la forma tradicional, on és el programador l'encarregat de solucionar els problemes de comunicació entre els dos móns: el relacional i el conceptual.

En aquest cas, el que hem hagut de fer és crear dins la capa comú, una classe per a cadascuna de les taules de la nostra BBDD, com es mostra a la imatge lateral.

Posteriorment, a la capa de dades, hem creat una classe ADO on hem hagut d'implementar tots els mètodes necessaris per a accedir i modificar les nostres taules.

Figura 22. Implementació tradicional sense frameworks ORM

3.6.2 Implementació d'Entity Framework

Un cop creat l'Entity Data Model (durant el procés d'instal·lació) l'hem hagut de configurar, agregant les associacions i/o relacions d'herència necessàries i parametritzant-les com sigui necessari. En el nostre cas només tenim associacions i hem hagut de configurar-ne la cardinalitat, com podem veure a la imatge inferior:

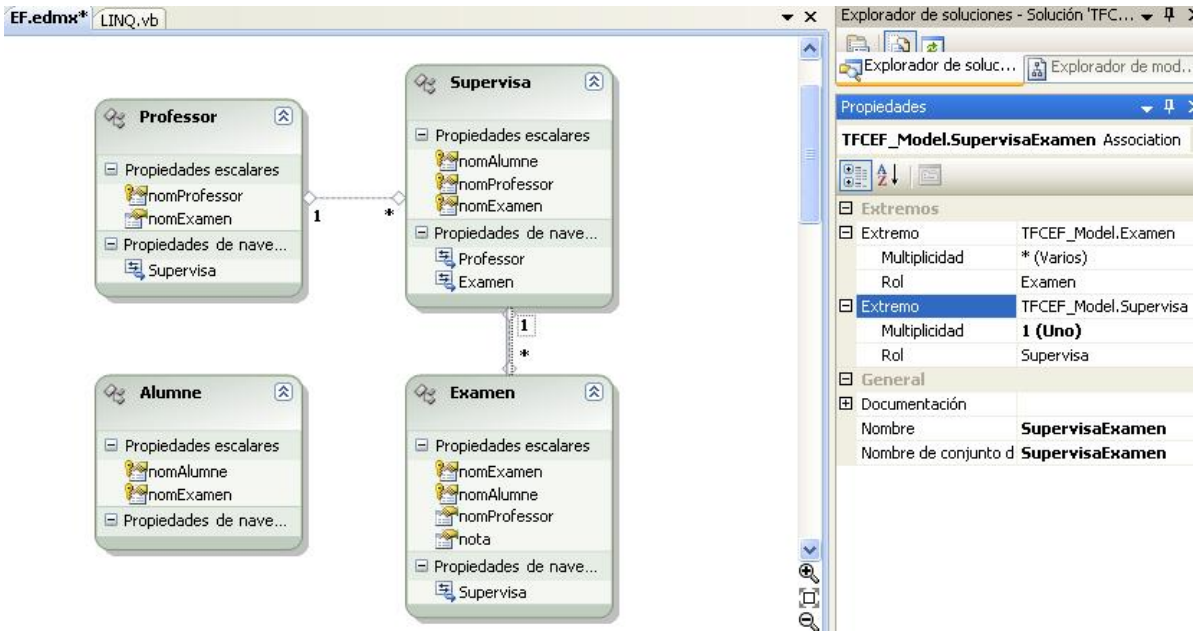
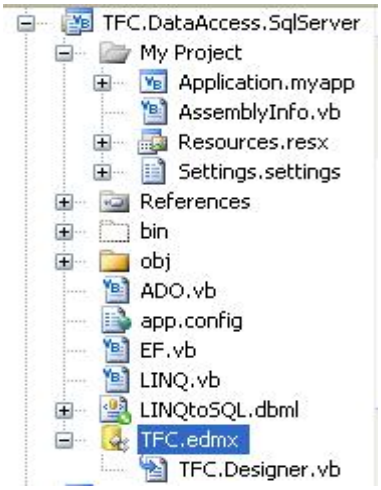


Figura 23. Configuració de les associacions i les seves cardinalitats

Tot seguit cal configurar també la taula de cada assignació, com es mostra:

Figura 24. Configuració de les taules de les associacions



El darrer pas és accedir al model creat des de la nostra aplicació. Això ho fem gràcies al model (TFC.edmx) que se'ns ha creat al projecte i mitjançant la classe EF que hem creat prèviament, on hem implementat els corresponents mètodes accessors getters/setters.

Val a dir que dins aquesta classe hem hagut d'importar les llibreries:

```
Imports System.Data.Objects
Imports System.Data.Objects.DataClasses

Així com accedir al model amb un objecte del
tipus del model:
Dim EF As New TFCEntities
```

Figura 25. Accés al model

Un cop feta la part teòrica sense cap incidència, hem d'apuntar no obstant, que hem tingut molts problemes en implementar l'Entity Framework i que finalment no hem aconseguit solucionar la següent excepció per moltes hores que hi hem dedicat i ajuda que hem demanat tant al fòrum de l'assignatura com en d'altres d'Internet:

http://social.msdn.microsoft.com/Forums/es-ES/vsgenerales/thread/b5bc9f1d-eba1-465a-a127-f3c1e1ff2e95

Figura 26. Excepció que no s'ha pogut solucionar

Com a darrera prova, hem intentat també generar el model per mig d'un segona opció que ja vàrem avançar al treball teòric de la PAC2: l'EdmGen.exe, una eina de línia de comandes usada per a treballar amb arxius de model i d'assignació d'Entity Framework, disponible a partir del Framework 3.5 i que permet generar un model conceptual simple.

El fet d'haver generat aquesta segona eina posteriorment ens fa ratificar més encara en la impressió de que aquest ORM és encara massa novell i problemàtic i probablement per aquest motiu es devia crear aquesta altra possibilitat de modelització.

Per implementar-la, el primer pas que hem hagut de fer és generar el model usant una comanda específica que lògicament ens hem hagut de configurar per a accedir a la nostra base de dades i solució en el format desitjat com es veu a la imatge inferior:

```
C:\WINDOWS\system32>"%windir%\Microsoft.NET\Framework\v3.5\edmgen.exe" /mode:ful
lgeneration /c:"Data Source=CASA; Initial Catalog=TFC; Integrated Security=SSPI"
 /project:TFC /entitycontainer:TFCEntitiesGen /namespace:TFC.DataAccess.SqlServe
r /language:VB
Microsoft (R) EdmGen versión 3.5.0.0
Copyright (C) 2008 Microsoft Corporation. Reservados todos los derechos.

Cargando información de la base de datos...
Escribiendo archivo ssdl...
Creando capa conceptual desde capa de almacenamiento...
Escribiendo archivo msl...
Escribiendo archivo csdl...
Escribiendo archivo de capa de objeto...
Escribiendo archivo de vistas...

Generación completada: 0 errores, 0 advertencias

C:\WINDOWS\system32>_
```

Figura 27. Generació dels arxius d'assignació i de model

Un cop executada la comanda de generació, aquesta crea els següents arxius, que no es veuen a la solució del Visual Studio:

```
C:\WINDOWS\system32>dir TFC.*
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 4C8B-4334

 Directorio de C:\WINDOWS\system32

12/12/2010  00:35             3.038 TFC.csdl
12/12/2010  00:35             2.035 TFC.msl
12/12/2010  00:35            28.334 TFC.ObjectLayer.vb
12/12/2010  00:35             6.471 TFC.ssdl
12/12/2010  00:35            20.526 TFC.Views.vb
               5 archivos         60.404 bytes
               0 dirs  47.263.932.416 bytes libres
```

Figura 28. Arxius generats

El segon pas és validar el model generat, amb una altre comanda específica i que també hem hagut de configurar-nos:

```
C:\WINDOWS\system32>"%windir%\Microsoft.NET\Framework\v3.5\edmgen.exe" /mode:Val
idateArtifacts /inssdl:.\TFC.ssdl /inmsl:.\TFC.msl /incsdl:.\TFC.csdl
Microsoft (R) EdmGen versión 3.5.0.0
Copyright (C) 2008 Microsoft Corporation. Reservados todos los derechos.


Validación completada: 0 errores, 0 advertencias
```

Figura 29.Validació del model

El tercer pas és generar un arxiu que contingui les classes d'objectes creades a partir de l'arxiu del model conceptual (*.csdl). Per això executem aquesta comanda:

```
C:\WINDOWS\system32>"%windir%\Microsoft.NET\Framework\v3.5\edmgen.exe" /mode:Ent
ityClassGeneration /incsdl:.\TFC.csdl /outobjectlayer:.\TFC.Objects.vb /language
:VB
Microsoft (R) EdmGen versión 3.5.0.0
Copyright (C) 2008 Microsoft Corporation. Reservados todos los derechos.

Escribiendo archivo de capa de objeto...

Generación completada: 0 errores, 0 advertencias
```

Figura 30. Generació del codi de nivell d'objecte

El quart pas és generar un arxiu Visual Basic (en el nostre cas) que contingui les vistes generades prèviament per al model existent. Això és fa des del l'IDE del VS anant a "propiedades del proyecto / compilar / generar eventos / evento anterior a la generación siguiente" i afegint el codi següent, que hem testejat anteriorment des del "cmd":

"%windir%\Microsoft.NET\Framework\v3.5\edmgen.exe" /nologo /language:VB /mode:ViewGeneration "/inssdl:C:\Documents and Settings\Sergi\Mis documentos\Visual Studio 2008\Projects\TFC\TFC.DataAccess.SqlServer\TFC.ssdl" "/incsdl:C:\Documents and Settings\Sergi\Mis documentos\Visual Studio 2008\Projects\TFC\TFC.DataAccess.SqlServer\TFC.csdl" "/inmsl:C:\Documents and Settings\Sergi\Mis documentos\Visual Studio 2008\Projects\TFC\TFC.DataAccess.SqlServer\TFC.msl" "/outviews:C:\Documents and Settings\Sergi\Mis documentos\Visual Studio 2008\Projects\TFC\TFC.DataAccess.SqlServer\TFC.Views.vb"

En aquesta imatge mostrem com ho hem fet des de l'IDE:

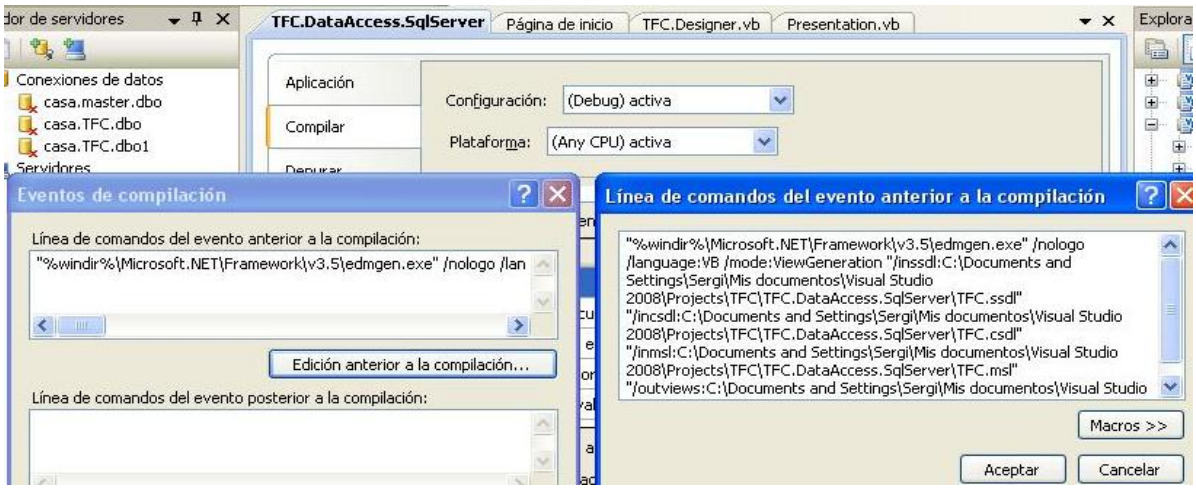


Figura 31. Afegir les vistes a la solució

Seguidament hem hagut d'agregar la vista al nostre projecte, fent un "agregar elemento existente", com mostrem:

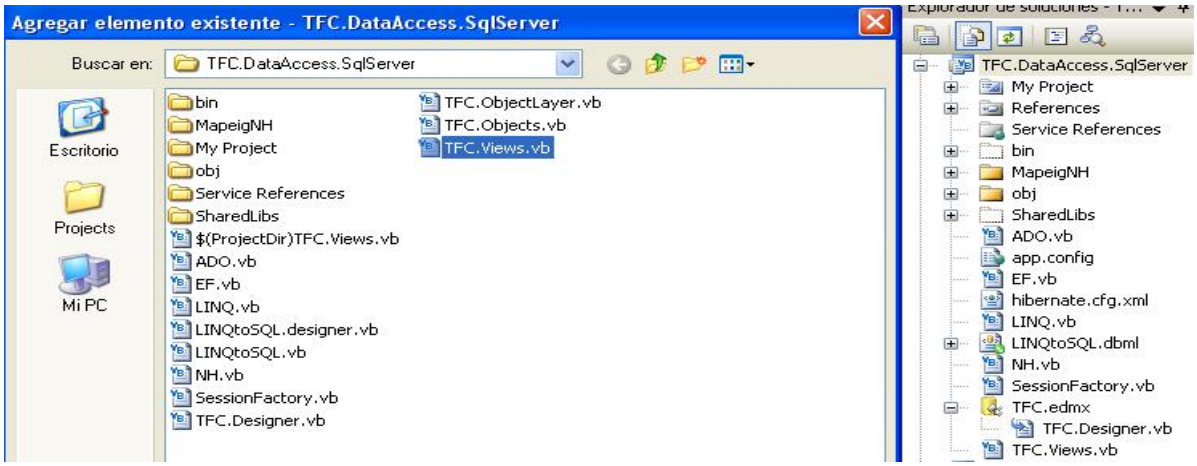


Figura 32. Agregació a la solució de les vistes generades

El mateix hem hagut de fer amb els arxius TFC.csdl, TFC.ssdl i TFC.msl, i en aquest cas a més a més, els hi hem hagut de configurar la propietat “Acción de compilación” com a “Recurso incrustado”. Novament inserim una imatge de les accions realitzades:

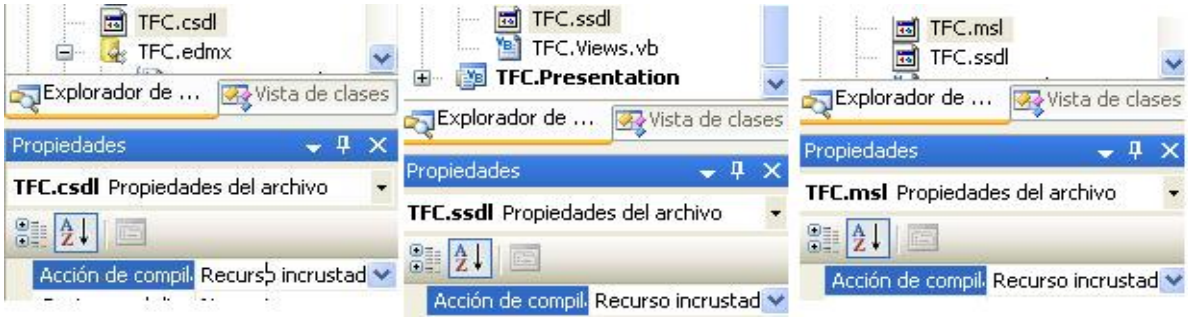


Figura 33. Agregació i configuració dels arxius TFC.csdl, TFC.ssdl i TFC.msl

Com a cinquè pas hem hagut d’obrir el fitxer “app.config” del nostre projecte i afegir-hi una nova cadena de connexió:

```
<add name="TFCEntitiesGen"
connectionString="metadata=res://*/TFC.csdl|res://*/TFC.ssdl|res://*/TFC.m
sl;provider=System.Data.SqlClient;provider connection string=&quot;Data
Source=CASA;Initial Catalog=TFC;Integrated
Security=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
```

Per acabar hem intentat accedir a la classe que ens hauria d’haver proporcionat les vistes d’accés a la nostra BBDD (TFC.Views.vb) però tampoc en aquest cas no hem estat capaços de fer-ho, degut un cop més a la pèssima documentació del MSDN relativa al EF, i la quasi total inexistència d’altres recursos formatius a la xarxa.

Ens ha quedat per tant la impressió de que l’Entity Framework és una tecnologia no massa implementada, poc madura i força problemàtica, que no recomanem en absolut, al menys en aquesta primera versió. Caldria comprovar si la segona, subministrada amb el Framework 4.0 i el VS 2010 es capaç o no de solucionar tots aquests problemes.

3.6.3 Implementació d'NHibernate

El primer que hem hagut de fer en aquest cas és agregar les referències a les llibreries d'NH, que estan dins la carpeta on hem descomprimit els fitxers de la instal·lació.

Després hem hagut de crear a la capa comú, objectes per a cadascuna de les taules relacionals. Un cop fet, el següent pas ha estat fer el mapeig. Per això hem creat una carpeta anomenada MapeigNH on hem afegit un arxiu XML per a cadascuna de les taules a comunicar, en el nostre cas n'hem creat un anomenat ExamenNH.hbm.xml, on hem generat el codi XML necessari per a mapejar les propietats de la classe. Generat el codi XML, dins la configuració del fitxer ha calgut configurar l'acció de compilació com a "Recurso Incrustado" perquè el fitxer s'incrusti a l'assemblat.

Seguidament hem hagut de generar una nova classe que hem anomenat SessionFactory, la qual és l'encarregada de llegir els fitxers de mapeig i configuració, i de mantenir una sessió que ens permeti la interactuació amb la BBDD. Dins la classe ha calgut posar alguns mètodes com "Init()", "GetSessionFactory()" o "GetNewSession()" que criden a funcions d'NH, per tant a la classe prèviament s'han hagut d'implementar els "imports" necessaris.

Fet això, ha calgut configurar NHibernate per a dir-li quina BBDD estem usant, que en el nostre cas és l'SQL Server 2008. Ha calgut doncs afegir-hi el següent codi:

```
<?xml version="1.0" encoding="utf-8" ?>

<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">

  <session-factory>

    <property name="connection.provider">
      NHibernate.Connection.DriverConnectionProvider
    </property>

    <property name="dialect">
      NHibernate.Dialect.MsSql2008Dialect
    </property>
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </property>

    <property name="connection.connection_string">
      Data Source=.\CASA;
      AttachDbFilename=|DataDirectory|\TFC;
      Integrated Security=True;
      User Instance=True;
    </property>

    <property name="proxyfactory.factory_class">
      NHibernate.ByteCode.Castle.ProxyFactoryFactory, NHibernate.ByteCode.Castle
    </property>

  </session-factory>

</hibernate-configuration>
```

Ara ja només ens manca accedir a les dades des de la nostra aplicació. Per una banda ha calgut crear una classe del tipus objecte de la BBDD, amb els corresponents atributs i mètodes necessaris, i per altra banda, hem hagut de definir el mapeig de cada classe a la que vulguem accedir. Per això hem creat un arxiu XML que s'anomena com el nom de la classe mes ".hbm.xml". Enganxo aquí el mapatge del fitxer ExamenNH.hbm.xml:

```
<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" namespace = "TFC.Common" assembly
= "TFC.DataAccess.SqlServer">

   <class name="ExamenNH" table="Examen" lazy="false">

   <id name="nomExamen">
     <generator class="guid" />
   </id>

      <property name="nomExamen">
    <column name="nomExamen" not-null="true" />
   </property>

   <property name="nomAlumne">
     <column name="nomAlumne" not-null="true" />
   </property>

   <property name="nomProfessor">
     <column name="nomProfessor" not-null="true" />
   </property>

   <property name="nota">
     <column name="nota" not-null="true" />
   </property>

 </class>

</hibernate-mapping>
```

Finalment però, amb NHibernate ens ha passat el mateix que amb l'Entity Framework. La teoria i la creació del model són acceptablement clares i entenedores (deixant de banda que cal tenir coneixements d'XML) però alhora de posar-ho en pràctica la cosa canvia. Tot i dedicar-hi incomptables hores i de cercar ajuda arreu on hem pogut, tant al fòrum de l'assignatura com especialment a la web i a fòrums d'Internet:

https://forum.hibernate.org/viewtopic.php?f=25&t=1008496&sid=9cd0b3596fed002395593e1ff5352a47

Novament no hem estat capaços de realitzar la connexió perquè no hem pogut solucionar la següent excepció:

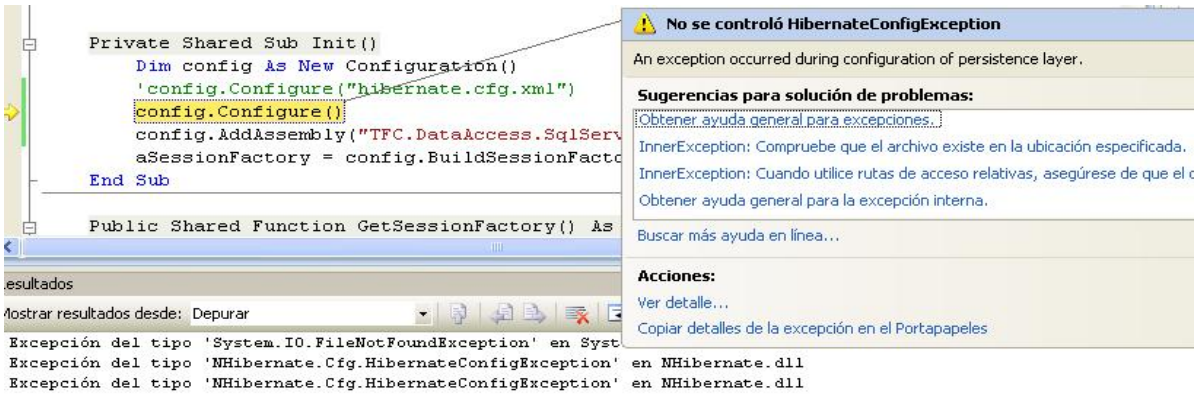


Figura 34. Excepció no solucionada

Ens ha quedat un altre cop la sensació de que aquesta també és una tecnologia força complexa i especialment molt poc implementada en l'entorn Microsoft.

3.6.4 Implementació de LINQ to SQL

Un cop agregada la classe LINQ to SQL (durant el procés d'instal·lació tractat a l'apartat 1.3.3) ens apareix una pantalla com la següent:

Figura 35. Interfície de l'ORD

El següent pas per a implementar el model està molt elaborat i és totalment gràfic (res a veure amb l'EdgGen.exe o l'NH) i consisteix en arrossegar a aquesta pantalla les taules de la nostra BBDD des de l'explorador de soluciones:

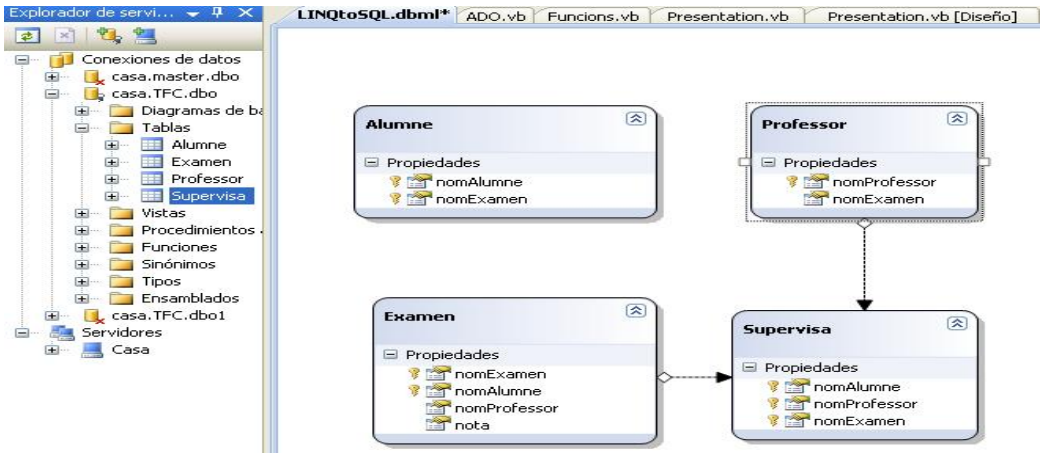


Figura 36. Creació de les classes simplement arrossegant-les des de l'explorador de servidores

Després en el cas de que tinguem herència o associacions, les hem de definir i ajustar la seva cardinalitat, que pot ser: oneToOne o OneToMany. A l'exemple de la imatge inferior es veu com ha quedat modelitzada l'associació ternària Supervisa. Concretament la imatge és del moment en que estàvem ajustant l'associació entre la classe associativa Supervisa i la classe Alumne, on es veu que hem configurat la cardinalitat com OneToMany.

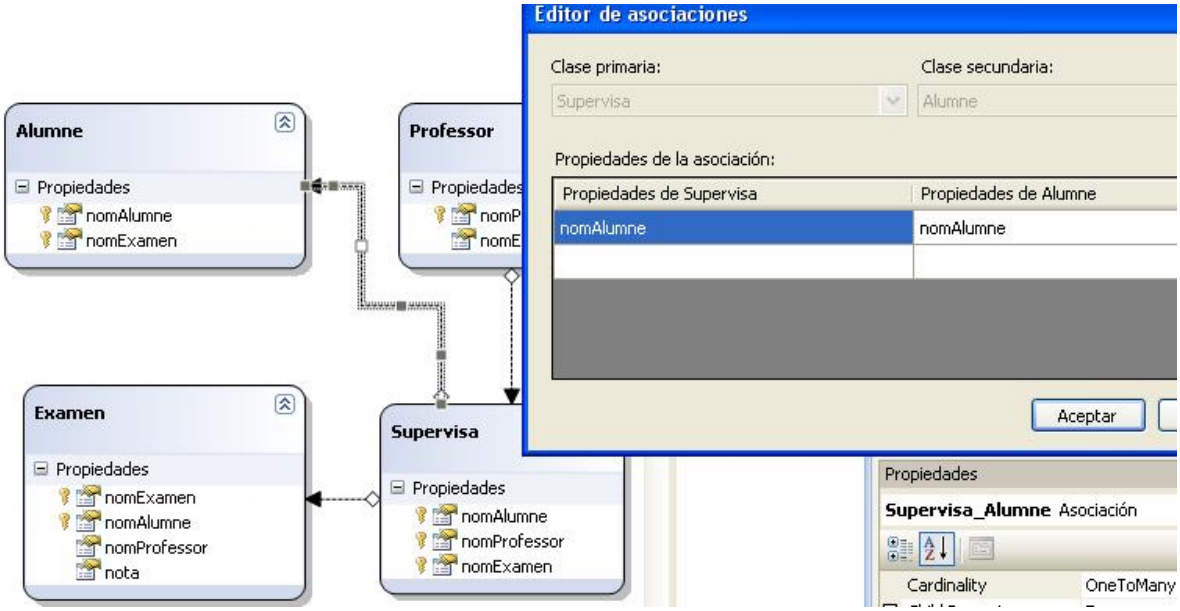


Figura 37. Modelització de l'associació ternària Supervisa

Després hem modelitzat la resta d'associacions del model: l'associació "Genera" d'un professor respecte a 1..* exàmens, i l'associació "Realitza" d'un alumne respecte a 1..* exàmens. Finalment el nostre model acabat ha quedat com es mostra a la imatge inferior:

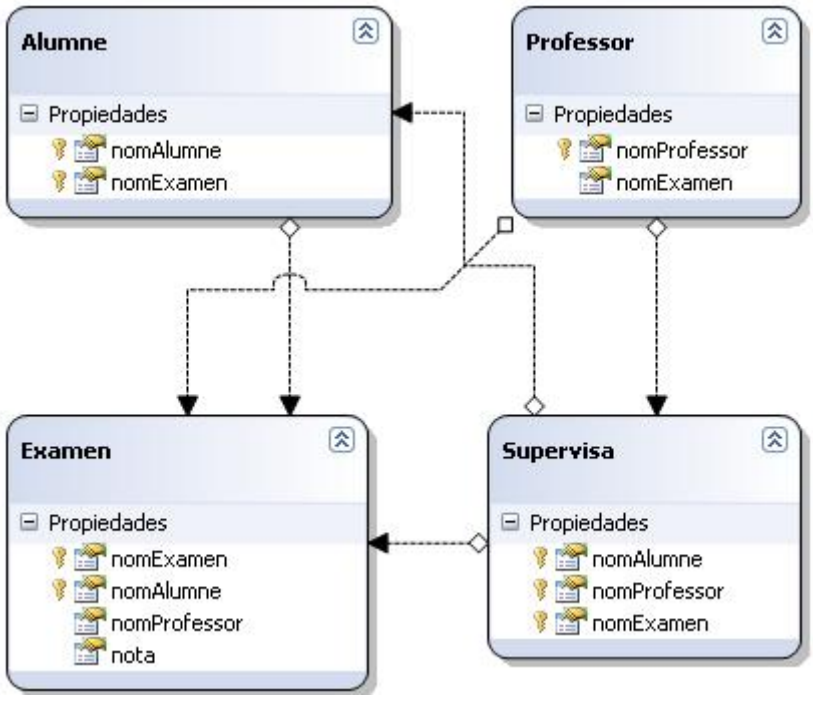
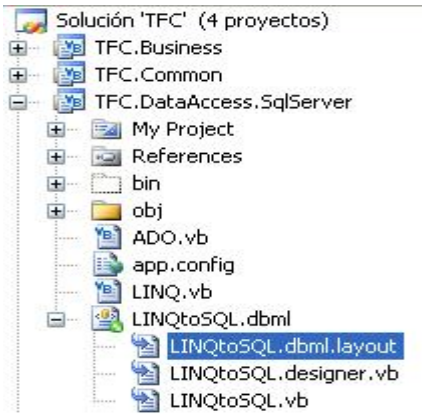


Figura 38. Diagrama de classes

El següent pas ha estat accedir al diagrama de classes des de la nostra aplicació.



Per això hem hagut d'accedir a la classe que se'ns ha creat al projecte quan hem fet l'arrossegament de les classes des de l'explorador de soluciones: LINQtoSQL.dbml i mitjançant l'objecte DataContext:

```
Dim LINQ As New LINQtoSQLDataContext
```

Aquest objecte és l'origen de totes les entitats assignades a una connexió de base de dades. Tot això ho hem fet a la classe LINQ que hem creat prèviament a la nostra capa d'accés a dades, classe on hem hagut d'implementar els corresponents mètodes accessors i modificadors.

Figura 39. Accés al model

Aquesta implementació també ha costat força, però no fins al límit dels dos ORM anteriors, ja que al menys la documentació sobre LINQ que hem trobat en aquest cas, tant a l'MSDN com a la xarxa, és força important i no té res a veure amb els altres dos frameworks. A més a més, hi ha força literatura disponible sobre aquesta tecnologia.

Els principals problemes que hem tingut han estat amb les associacions i amb el fet de tenir una clau primària doble a la taula Examen, però un cop solucionats si que hem pogut en aquest cas realitzar la connexió al servidor de BBDD i realitzar la resta de casos d'ús.

L'únic punt que no hem estat capaços de solucionar amb el mètode tradicional és la funcionalitat d'afegir, la típica comanda "update" d'SQL, on ens han saltat les següents excepcions:

```
Excepción del tipo 'System.Data.SqlClient.SqlException' en System.Data.Linq.dll

Excepción del tipo 'System.Data.SqlClient.SqlException' en TFC.DataAccess.SqlServer.dll
```

Tot i que novament en aquest cas hi hem dedicat moltes hores i de que hem trucat a totes les portes que hem pogut buscant ajuda externa, tant al fòrum de l'assignatura com a Internet:

http://social.msdn.microsoft.com/Forums/es-ES/vsgenerales/thread/125c0786-86c1-4ee0-955b-a75fc237924a

Finalment però, i gràcies a l'extensa literatura existent, hem pogut executar aquesta funcionalitat encara que seguint un altre via: la implementació de procediments emmagatzemats a la base de dades.

Val a dir que tant de bo haguéssim descobert abans els "stored procedures", perquè aquesta és una forma de treballar molt més fàcil que no pas el llenguatge específic del LINQ. En aquest cas tot són comandes estàndard d'SQL, que a més ens han funcionat a la primera.

3.6.5 Breu resum de les diferents implementacions

Després d'haver estudiat i treballat de forma intensiva les quatre tecnologies, la impressió que ens ha quedat és la següent.

ADO.NET és la tecnologia tradicional, la més estesa encara hores d'ara i la més senzilla de totes. El motiu és fàcil d'entendre: a més a més de que és una tecnologia molt més madura amb forces anys d'experiència a darrera, esta basada en el model relacional i no pas en el conceptual, per tant permet un accés directe a les dades ja que no es preocupa de cap conversió entre els móns conceptual i relacional.

No obstant, aquesta tecnologia l'hem implementada només per poder-la comparar amb les altres tres i no era pas objecte d'estudi d'aquest treball. A més, avui dia en que el gran nombre d'aplicacions es desenvolupen sota el paradigma de la programació OO, cada cop té més sentit que el programador es despreocupi de la diferencia entre els dos móns.

Entrant ja en els diferents frameworks ORM, sens dubte el menys complex i més amigable amb diferència de treballar és el LINQ to SQL, i especialment accedint a la persistència amb els procediments emmagatzemats, via que estalvia haver de conèixer en detall el llenguatge de LINQ, doncs en aquest cas es treballa directament amb l'estàndard SQL. Llàstima que vàrem descobrir tard aquesta tecnologia perquè si l'haguéssim implementat des d'un bon principi ens hagués costat molt menys de posar en marxa.

Aquesta és doncs la tecnologia que usaríem sens cap tipus de dubte si haguéssim de desenvolupar una aplicació amb un nombre d'entitats important i haguéssim de implementar la persistència en una base de dades relacional.

Entity Framework, tot i ser també de Microsoft, és encara una tecnologia massa novell i molt poc implementada, no hi ha quasi gens d'informació a Internet fora del MSDN, i aquesta és escassa, complexa i difícil d'interpretar de forma autodidacta. Em provat les dues alternatives vàlides per a generar el model EDM i les hem pogut crear les dues, però en la vessant pràctica, en cap dels dos casos hem aconseguit connectar amb la BBDD tot i cercar ajuda arreu per Internet en fòrums especialitzats, dels quals no hem rebut cap resposta. Ha estat una experiència bastant depriment i gens recomanable.

NHibernate no surt massa més ben parat. Desconeixem si realment està a l'alçada del seu germà gran Hibernate, tecnologia força implementada en el món Java, però ens ha donat la impressió de que al igual que l'EF, aquesta és també una tecnologia molt poc estesa en l'entorn de programació de Microsoft. Costa de trobar informació a la xarxa (encara que ni ha més que no pas d'EF) i la que hi ha si que és suficient per a generar el model, cosa que també hem aconseguit, però no pas per a posar-lo en marxa, tot i que en aquest cas també hem intentat cercar ajuda als fòrums especialitzats sense tampoc cap èxit. Novament aquesta ha estat una experiència molt frustrant que tampoc recomanem en absolut.

3.7 Arquitectura

Hi ha una part comú a tota l'aplicació, i una part variable, la de la capa de dades, que depèn de cadascuna de les tres aplicacions i que vàrem explicar en detall a la PAC2.

L'aplicació segueix el model de desenvolupament en tres capes. Aquest model és molt beneficiós, sobretot al treballar amb grups de treball. Intenta separar les tres grans àrees de l'aplicatiu en tres blocs, afavorint la reutilització. Aquestes són:

1. **Capa de presentació**, interfície entre l'usuari i l'aplicació.
2. **Capa de negoci**, on hi trobarem les diferents funcionalitats del negoci.
3. **Capa de dades**, encarregada de l'accés a dades i de gestionar la persistència. Aquesta és la capa més complexa en el nostre cas, degut a les quatre tecnologies a desenvolupar.

Per a implementar el model de 3 capes en Visual Studio, cadascuna d'elles s'ha de posar en un projecte. La capa de presentació és un projecte de tipus Windows Form en el nostre cas, i la resta són projectes de biblioteques de classes. Per a lligar els projectes entre si, ho hem hagut de fer mitjançant "Agregar referencia \ proyecto", per a cadascun d'ells.

A la figura inferior mostrem l'arquitectura de l'aplicació desenvolupada amb aquest model.

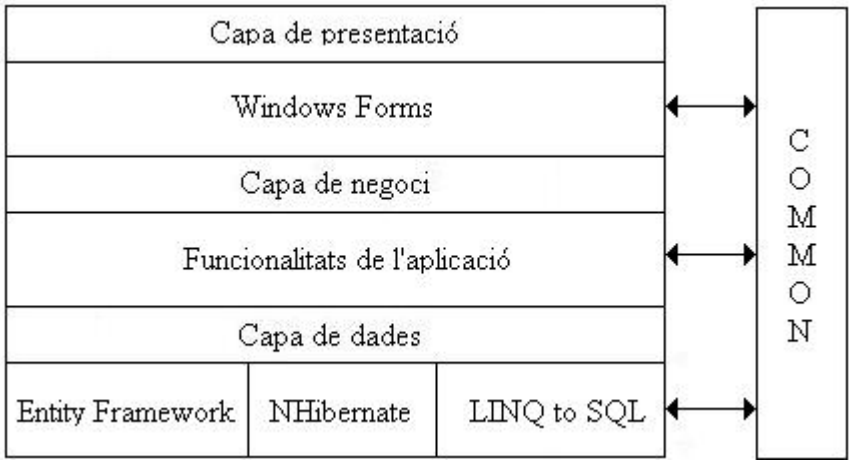


Figura 40. Arquitectura en 3 capes

4 Conclusions

4.1 Objectius aconseguits

El principal objectiu de la selecció d'aquesta àrea del TFC era el coneixement de l'entorn de programació Microsoft .NET, una de les principals plataformes de programació que hi ha actualment junt amb Java. Aquest darrer ja el coneixíem força degut a que és una eina usada àmpliament en aquests estudis, ans al contrari de .NET, que és una plataforma propietària però amb molt demanda als mercats.

Un cop seleccionada l'àrea d'interès el següent pas va estar escollir el treball més adequat i vàrem triar el de les tecnologies d'accés a dades per a conèixer a fons algunes de les tecnologies més importants en quelcom tant bàsic per a qualsevol aplicació com és la implementació de la persistència.

Quan vàrem seleccionar aquest treball no coneixíem encara el concepte d'ORM i ha estat una grata experiència descobrir que ja és possible implementar la persistència pensant únicament en el model conceptual i oblidant-nos de la dificultat de la conversió entre aquest món i el relacional.

La primera etapa d'estudi i recerca va ser molt il·lusionant, descobrint les principals tecnologies que implementen aquest model, però alhora de posar en pràctica la teoria vàrem haver de patir força i dos d'aquests frameworks ens han resultat especialment frustrants al no haver aconseguit solucionar totes i cadascuna de les moltes excepcions de programari aparegudes, per moltes hores que hi hem dedicant fent diverses proves i buscant informació i suport arreu, tant en literatura específica com especialment a la xarxa.

Tot i així, entenem que qualsevol nova tecnologia necessita el seu temps de rodatge abans de consolidar-se i probablement amb més temps les dues que ens han donat més problemes (EF i NH) acabaran essent més suportades i acceptades, sempre que Microsoft hi posi de la seva part en el cas de la tecnologia de codi obert (NH).

Per la nostra banda, aquest treball ha complert satisfactòriament tots els objectius previstos. Hem assolit un bon coneixement de la plataforma .NET, de l'IDE Visual Studio 2008, del SGBD SQL Server 2008 i de les tecnologies: Windows Forms, Visual Basic.NET, ADO.NET, Entity Framework, NHibernate, LINQ i LINQ to SQL, a més d'haver refrescat altres tecnologies com l'XML o l'SQL.

4.2 Treball futur

En un futur seria molt interessant veure com evolucionen els diferents frameworks ORM. Avui en dia, LINQ to SQL sembla que és amb diferència el millor dels estudiats, tant per la seva interfície visual de creació del model, com per l'extensa literatura associada i la seva acceptació, doncs només cal buscar qualsevol tema relacionat a la xarxa per veure la quantitat de dades que ens retorna el navegador, cosa que no passa amb els altres dos.

Microsoft disposa de dues alternatives per a implementar el model: LINQ to SQL i Entity Framework. Sobre aquesta darrera, hem de dir que hem usat i avaluat la primera versió del producte, caldria estudiar com és la segona apareguda amb el Framework 4.0. L'avantatge de LINQ sobre aquesta primera versió d'EF es nota en molts aspectes: la potència visual de creació del model del LINQ, l'amplia documentació associada, etc. Tot el contrari de l'EF, cosa per altra banda relativament normal en primeres versions.

Per la seva banda, NHibernate és un producte de codi obert, pensat per a portar la tecnologia Hibernate que triomfa en Java a la plataforma .NET, però després de l'estudi dubtem de que aquesta solució interessi a Microsoft, ja que no dona la impressió de que ajudi massa a que aquest projecte tiri endavant.

En qualsevol cas, serà molt interessant estar al dia de com evolucionen aquestes tecnologies, doncs si tenim en compte que:

1. la gran majoria d'aplicacions avui dia es desenvolupen orientades a objectes,
2. els SGBD no relacionals segueixen sense tenir una gran acceptació al mercat,
3. el model ORM es l'única alternativa actual que permet alliberar als programadors de la dificultat de comunicació entre les aplicacions conceptuals i la persistència relacional.

Sembla clar que aquest marc té encara molt de futur i molt de camí a recórrer, així que haurem d'estar atents i seguir la seva evolució.

4.3 Resum i reflexió final

En aquest darrer apartat del treball i per primera vegada en tot el projecte, parlaré en primera persona per expressar encara de forma més clara el que ha significat personalment per a mi.

Aquest és el segon treball final de carrera que faig. El primer el vaig fer quan vaig cursar ETIS i va ser un projecte molt més teòric, dedicat als aspectes de disseny i usabilitat de les interfícies home màquina (HMI).

Aquest segon treball volia que fos tot el contrari, un projecte més pràctic que em permetés conèixer una de les principals interfícies de desenvolupament integrat d'elaboració de programari actuals. Vaig estar dubtant entre les àrees de Java2EE i .NET, i finalment em vaig decidir per aquest darrer perquè en el món on jo treballo: l'automàtica industrial, estan molt més implantades les solucions Microsoft.

Val a dir que durant el semestre passat vaig fer un curset introductori de .NET per a preparar-me aquest TFC, però aquesta plataforma agrupa i integra tantes tecnologies (VB, C#, ADO, WF, WPF, ASP, LINQ, AJAX, Reporting, Reflection, Silverlight...) que és molt difícil dominar-les totes. Un clar exemple és que fins després de començar el treball no havia sentit a parlar mai del concepte d'ORM ni de les tecnologies EF, NH i LINQ to SQL.

Tot i així, amb els fonaments que vaig aprendre durant aquest curs, més especialment tots el que he anat adquirint durant la implementació d'aquest projecte final de carrera, penso que he complert més que sobràdament el meu objectiu de conèixer la plataforma .NET, encara

que reconec que és un TFC al que cal dedicar moltes hores i constància degut a les diverses tecnologies que l'integren.

Estic per tant molt satisfet amb la feina feta i amb els coneixements que n'he tret. No ha estat gens fàcil ja que m'he trobat amb que alguns dels ORM seleccionats estaven molt poc implantats i quasi no he pogut trobar suport enlloc, ni al fòrum de l'aula (ja que no ha estat un treball escollit per altres companys) ni a la xarxa, on d'EF i NH he trobat suficient informació per implementar el model, però una ínfima quantitat alhora de buscar solucions a les meves excepcions.

No obstant penso que ha valgut la pena l'esforç i vull agrair l'ajut psicològic del meu consultor, que sempre m'ha animat a tirar endavant el projecte i a no descoratjar-me si no trobava cap sortida.